FreeRange Computer Design

Version: v11.00

Table of Contents

Pretentions

(Bryan Mealy 2016 ©)


Legal Crap

You can download a free electronic version of this book from one of (and only one) of the following sites:

http://www.freerangefactory.org

my calpoly teacher website

We are more than happy to consider your contribution in improving, extending or correcting any part of this book. For any communication or feedback that you might have regarding the content of this book, feel free to contact the author at the following address:

freerangecomputerdesign@gmail.com


Acknowledgements

Someday, I'll write something here.

Hey Dickson… Someday we'll work together on all the things we've yet completed. I'm looking forward to that day.

Rambling Commentary

My inspiration for this project came from my own personal notion that knowledge, particularly technical knowledge, should not be held ransom by publishing companies, bookstores, book authors, and academic administrators. Students seeking knowledge are sitting ducks when it comes to the notion of structured learning situations such as colleges and universities. Being that students are the lowest hanging fruit, they always are the first to have their wallets lightened by various well-connected entities. I hope this book serves as an alternative to shelling out money for overpriced textbooks.

This book is going to have errors. Please accept my sincerest apologies for the errors you will come across. I did my best to remove errors, but there are two main factors that mitigated my error removing initiative.

1. Writing and proofreading is very timing consuming.

2. Unlike several of my colleagues, I do not bribe students into proof-reading my writing. There has been more than one instance of an instructor at the institution where I teach giving "extra credit" to anyone who reported errors in their writing. I do happily accept suggestions and corrections from students, but it is out of the student's own desire to help on the project.

3. Every digital design problem in this book was generated by me. Once again, unlike many of my colleagues, I did not "assign" students to generate problems for me. I believe instructors who force students to create problems as graded assignments are unethical and are basically taking advantage of their positions are instructors. It's called an abuse of power. I simply won't do that.

I could spend the remainder of my life tweaking this text, but I need to move onto other things. By all means, feel free to contact me with corrections and comments. Please don't write me at my ".edu" address as that account is generally filled with pointless academic drivel (which I tend to ignore) or email from my students (which I generally attend to with a high priority). Please use this address to write me:

FreeRangeComputerDesign@gmail.com.

There were two primary negative comments I received when I mentioned I was writing a textbook and was planning to give it away at no cost.

- "If you don't charge something, people will not value it". I don't understand this statement. The things I value most in my life were given to me. Maybe I'm missing something here.

- "You need to have experts in your field review your text". As a college teacher, I constantly receive requests from book companies to "review" one of their texts. They always sweeten the deal with an offer of cash. I know of no one who is going to dedicate any significant amount of their time to reading a text they care nothing about, but I know of people who pretend to review books, write down some drivel, and receive their cash. Wow! Great review! A book is a mechanism to transfer knowledge; it's not a popularity contest, as are most things in academia.

As you read this book, you may get the impression that I don't like academic administrators. The truth is that I do not like academic administrators. In addition to cleaning out the wallets of students, they seem to want to rip the hearts out of students and teachers alike. They lack ethics and morals (an understatement). They reward those who support their agendas and crap on everyone else. They make sure they are first at the feeding trough and leave little for anyone else. They run the school as a business and not as an institutions of learning. They base every decision they make on economics; quality of education and the basic needs of students are never one of their primary considerations. Academic administrators seem to believe that students and teachers should be serving them; I believe we should all be serving students. Schools exist to help students learn; every decision should be made to support student learning. Academic administrators have clearly lost sight of the basic tenets of education.

Finally, this text is what it is. The quality and coverage is the best I can do given the various constraints I'm working under. I made the decision to embark on this project knowing that it was more than likely a career killer in the context of Cal Poly SLO. Well, no need to wonder anymore; it's definitely a career killer. I don't feel this would have been the case had it not been for the total lack of professionality by my esteemed colleagues in the Electrical Engineering Department. These people saw the writing on the wall and made the decisions to put their time and efforts into projects that supported their career aspirations rather than supporting students and the learning process in general. The academic environment in general encourages faculty to adopt characteristics of NPD (narcissistic personality disorder) in order to be labeled "successful"; part of that disorder is to find corporately acceptable labels for their rampant professional jealously. I chose to not compromise my ethics and not lose sight of the mission of an instructor, certainly a choice that people should not make if they operate with a complete lack of integrity.

Overview of Chapter Overviews

This text presents introductory computer design and assembly language programming concepts. This book focuses on a single generic computer design: The RAT Microcontroller. There are definitely topics in computer architecture and assembly language programming that this book does not cover. The general idea behind this textbook is to support a college course that attempts to teach computer design and assembly language programming in a ten-week course. While this barely seems possible, this textbook and various supporting materials do their best attempt to make it possible.

NOTE: Chapters 4, 5, and 21 appear in FreeRange Digital Design; they are repeated here to provide the complete hardware design packet for students interested in computer design fundamentals.

PART ONE: Introduction and Review

Part One of FreeRange Computer Design introduces the various aspects of computer design. This introduction includes a high-level overview of computers using terms and concepts that you would find in a typical digital design course. In addition, if you can't remember those terms and concepts, this part also provides a fast overview of the important digital concepts.

Chapter 1:
This chapter presents a relatively quick overview of both computer design and the approach this textbook takes to introducing computer design and assembly language programming. This chapter includes an overview of the course, a brief history of the course, and an introduction to the RAT Microcontroller. *This chapter is important because it provides a context for this text by describing some the issues regarding computer design courses as well as other pertinent information.*

Chapter 2:
This chapter introduces the notion computer design using terms associated with a beginning digital design course. The basic computer concepts include basic computer architecture, computer instructions, the programmer's model, the instruction set, and the various levels available to program computers. *This chapter is important because it provides a high-level overview of the course in terms that you should already know. This overview provides a roadmap of the course and the text.*

Chapter 3:
This chapter provides a review of the basic building blocks (modules) of digital design. These building blocks include combinatorial and sequential circuits, as well as Finite State Machines (FSMs). This chapter also provides a review of the important topics and characteristics involving VHDL, which we typically use as a way to model, synthesize and simulate digital circuits. *This chapter is important because it describes most of the important concepts from a typical beginning digital design course. In particular, this chapter provides a fast overview of the topics presented in FreeRange Digital Design.*

PART TWO: Hardware Foundations of Computer Design

Part Two of FreeRange Computer Design introduces the various hardware aspects of computer design that typical digital design courses often do not present. We present these topics because they are critical to understanding the low-level aspects of subsystems typically found in computers.

Chapter 4:
This chapter provides an overview of basic registers as well as some specialized registers commonly seen in digital circuits. We model the basic register as an "n" D flip-flops as the store elements. This chapter covers the basics regarding specialized registers such as tri-state register and bi-directional registers. *This chapter is important because registers and their simple variations are extremely useful and thus often found in just about all meaningful digital designs.*

Chapter 5:
This chapter introduces and a relatively complete coverage of two of the most common and useful type of registers: shift registers and counter. This chapter covers the many types of shift registers including barrel shifters, universal shift registers, etc. This chapter also covers the various types of counters such as up/down counters and decade counter. This chapter discusses counter implementation primarily at a behavioral level and contain relatively large amount of VHDL modeling. *This chapter is important because shift registers and counters are extremely useful in many areas of digital design, particularly in applications requiring fast arithmetic operations. These devices are simple registers with extended features.*

Chapter 6:
This chapter introduces Register Transfer Notation, which is highly useful in both designing and describing circuits. This chapter uses RTN to describe and design various classifications of data transfer circuitry commonly used in digital design. *This chapter is important because register transfer notation is highly useful in designing and/or describing computer operations because it provides a compact form to describe data transfers and the signals that control them.*

Chapter 7:
This chapter provides an introduction many of the more common and important aspects of structured memory. This chapter primarily involves the high-level characteristics of structured memory and describes them primarily in general terms. This memory introduction includes items such as standard memory vernacular and basic performance characteristics. *This chapter is important because it provides a basic overview of digital memory and the operation of memory in a digital system.*

Chapter 8:
This chapter introduces the basic concepts and vernacular regarding structured memory including memory mapping, memory organization techniques. This chapter also provides basic VHDL models for the a few common memory types generally founding in digital computer circuitry. *This chapter is important because it provides a basic overview of structured memory and various details of modeling and interfacing with that memory.*

Chapter 9:
This chapter provides an overview computer architecture as motivation for its discussion of Arithmetic Logic Units (ALUs). This chapter takes two approaches to ALU design, a low-level approach using standard digital modules, and a high-level approach using relatively advanced VHDL modeling. *This chapter is important because it describes several approaches to designing ALUs. This description includes an introduction to the use of variables in VHDL.*

PART THREE: Assembly Language Programming Background and Concepts

Part Three of FreeRange Computer Design provides the knowledge you'll need to become a RAT MCU assembly language programmer. We wrote this section of the book in such a way as to not include hardware details in the various programming topics in the chapter. We have instead opted to save the hardware details until the Part Four of the text. This will require repeating some of the concepts, but we make all efforts to keep the repetition to a minimum.

Chapter 10:
This chapter introduces many of the concepts surrounding the notion of computer programming. This chapter assumes the reader has some programming experience but does not assume any particular knowledge of assembly language programming. In essence, this chapter reminds the reader what they already know about computer programming as an introduction to assembly language programming. *This chapter is important because it introduces assembly languages and associated concepts without requiring any prior knowledge of assembly languages.*

Chapter 11:
This chapter introduces assembly languages, and in particular, the RAT assembly language. This introduction regards the underlying low-level details of the instructions and particularly the instruction set architecture (ISA) in terms that someone who knows nothing about assembly language can understand. The information in this chapter will become even more useful in later chapters. *This chapter is important because it provides a background regarding the assembly languages and particularly assembly language instruction set architectures (ISAs).*

Chapter 12:
This chapter introduces the main concepts of writing assembly language programs by describing their basic parts and by differentiating assembly language programs from typical computer programs written in introductory computer programming courses. This chapter includes several simple but complete assembly language programming examples. *This chapter is important because it describes the basic structure of assembly language programs and provides several well-comment assembly language example programs*

Chapter 13:
This chapter provides an introduction to some of the most important assembly language programming constructs in the context of the RAT MCU architecture. The chapter includes introductions to instructions utilizing scratch RAM and stack memory. This chapter also introduces some of the standard approaches to tweaking bits in a byte-oriented machine. *This chapter is important because it describes some of the basic programming issues involved in programming the RAT MCU and assembly language programming in general.*

Chapter 14:
This chapter describes some of the basic concepts and terminology associated with assembly language program. This chapter includes descriptions of how to efficiently complete various using what we often refer to as "tricks". In other words, this chapter describes some of the more important considerations programmers should be aware of when writing robust assembly language code. *This chapter is important because it describes some of the basic programming concepts and approaches beyond simple description of individual instructions.*

Chapter 15:
This chapter provides a large set of assembly language program and their commented solutions. The problems start out with simple programs and end with some moderately complicated problems. *This*

chapter is important because it shows how to solve a wide set of problems by writing RAT assembly language programs.

Chapter 16:
This chapter provides an overview of the RAT Assembler. The topics in this chapter include an overview of the automatically generated files, assembly directives, memory segmentation, assembly language program form, and the start-up code. *This chapter is important because it describes all aspects of the RAT assembler and how the RAT assembly language programmer can properly utilize them.*

Chapter 17:
This chapter introduces assembly language program design and structure through the use of flowcharting. Flowcharts provide a visual aid to understanding algorithms and program flow. Flowcharts also provide an excellent documentation tool. *This chapter is important because it describes the advantages of using flowcharts in assembly language program design and documentation.*

PART FOUR: RAT MCU Architectural Details

Part Four of FreeRange Digital Design provides the hardware details regarding the RAT MCU architecture. This part also includes information on modifying the RAT MCU as well as interfacing the RAT MCU to external peripheral devices.

Chapter 18:
This chapter describes the various modules in the RAT MCU architecture. We make an effort to describe these modules on both a local and system level. This chapter is young and wild; it needs some better direction. *This chapter is important because it describes the low-level architecture details of the RAT MCU on both a local and system-level.*

Chapter 19:
This chapter provides more details involved in understanding and using the RAT MCU in actual implementations. This chapter includes an in-depth timing analysis of RAT MCU instructions, a description of the infamous “RAT MCU Wrapper”, a description of RISC & CISC architectures, and a discussion of levels of memory as it applies to the RAT MCU. *This chapter is important because it describes some the non-architectural but still important details involving the RAT MCU.*

Chapter 20:
This chapter provides descriptions of modifications and extensions to the RAT MCU. The modifications in this chapter are designed to extend the overall knowledge and understanding of the RAT MCU, with the idea being that you can understand the RAT architecture if you can describe meaningful and efficient changes to it. *This chapter is important because it advances your knowledge of the RAT MCU by outlining possible hardware architecture changes in response to stated design goals.*

Chapter 21:
This chapter provides an overview of interfacing to external peripheral devices to the RAT MCU. This chapter provides example for simple memory and ADC devices that use a simple set of interface design requirements presented in the chapter. *This chapter is important because it provides some low-level details of interfacing to common electronic devices.*

Chapter 22:
This chapter provides an overview of testbenches in VHDL. We include this section because testbenches are the primary tool that designers use to test their design. This is a basic introduction; there is nothing overly fancy in this chapter. *This chapter is important because it provides an overview and introduction to writing testbenches in VHDL. The VHDL language uses testbenches as a mechanism for verifying the proper operation of VHDL models using other VHDL models.*

Appendix
The Appendix provides some useful and handy RAT MCU information as well as fast overviews of VHDL. These items include:

- RAT MCU Architecture and Assembly Language Cheat Sheet
- VHDL Cheat Sheet
- Finite State Machine Modeling using VHDL Behavioral Models
- RAT MCU Architectural Diagrams
- RAT MCU Wrapper Source Code
- VHDL Style File
- RAT MCU Assembly Language Style File
- Requiem for the Digital Designer

Computer Design Dictionary
This item provides a list of important computer design terminology and their relatively brief definitions.

Index
This item provides fast locator for the more important terms and acronyms used throughout the text.

PART ONE: Introduction and Review

1 Chapter One: FreeRange Computer Design Overview

1.1 Introduction

The main purpose of this chapter is to put FreeRange Computer Design into a meaningful context. I'm hoping to give you this context in several ways: 1) by describing the outline of the various chapters in this text, 2) by describing some of the issues involved with "computer design" textbooks, 3) by providing you with a general overview of the course, and finally, 4) by providing an quick history of the course. While all of this is not killer useful information, having the proper context for your endeavors will facilitate learning, which is never a bad thing.

Main Chapter Topics

- **DESCRIPTION OF CHAPTER FORMATS:** Each chapter has a similar format; this chapter describes the chapter format for FreeRange Computer Design.
- **OVERVIEW OF TEACHING COMPUTER DESIGN:** Computer Design means different things to different people; this chapter describes the relatively unique approach for FreeRange Computer Design.
- **TEXT AND COURSE HISTORY**: This text and course have had a relatively long history. This chapter mentions some of the finer points and acknowledges the people who did the work to make this course text happen and continually improve.

Why This Chapter is Important

This chapter is important because it provides a context for this text by describing some the issues regarding computer design courses as well as other pertinent information.

1.2 Chapter Structure

Each chapter has many useful features in order to help the reader organize and digest the material in the chapter. Each chapter generally has the following features, though some chapters have special formats of their own.

- **Introduction**: Quick motivating prose overview including a list of the main topics and the chapter and why that chapter is important in digital design
- **The Body of the Chapter**: In case you want the whole story (with example problems)
- **Chapter Summary**: The quick overview of chapter
- **List of New and Important Terms**: Another quick summary notion
- **Practice Problems**: Including both exercises and/or design problems for the reader's entertainment

1.3 FreeRange Computer Design Beginnings

This text presents a course in Computer Design and Assembly Language Programming. The original label for this course was CPE 229 (the lecture portion of the course) and CPE 269 (the laboratory portion of this course). We later changed the course delivery to a studio format, which also entailed a label switch to CPE 233. Cal Poly first taught the original CPE 229/269 course-set Fall 2003 quarter. The current version of CPE 233[1] has developed considerably in the past few years and has become one of the most worthwhile courses offered by the Electrical Engineering Department. CPE 233 is still a young course; unlike many course offered in the EE Department, CPE 233 is under constant improvement and is well on its way to becoming a great course.

Several years ago, a bunch of old guys[2] sitting around a table dreamed up a change in the CSC, CPE, and EE curriculum that was perceived to improve the quality of education in the respective departments. The idea was to do-away with EE 319 (hardware-based finite state machines and advanced digital design) and CSC 215 (software-based 68000 assembly language programming) and compress those topics into a single course. In reality, both of these courses went relatively deep into their respective topics. As if this was not enough, they also decided to add an element of computer architecture to the course. The initial result was a highly specialized course that covered finite state machine design, basic computer architecture, and assembly language programming. We later removed the notion of finite state machine design and placed it into the beginning digital design course (CPE 133). FSM design is an important topic to computer design, so this course retains some important aspects of FSM and intermediate level digital design.

This change in curriculum created several problems, some of which we are still dealing with today. Here are the gory details and status of these problems:

1. The first problem is there is no existing book that is appropriate for the entire course. Unfortunately, this has resulted in a compromised learning experience for the students taking the class. The instructor that spearheaded the development of CPE 229/269 originally promised to provide teaching materials for the instructors teaching the course, but the materials provided were not only worthless, but an on-going joke[3] amongst the students taking the course. In truth, this instructor's primary focus was to use CPE 229/269 as a vehicle for him to write another useless textbook and subsequently force CPE 229/269 students to purchase the text. This instructor finally retired. He did finish the book, however; no surprise that no Cal Poly instructor ever used his book after his retirement, which stands as a testament to the overall quality of this instructor's product and professionality.

2. The second problem that this curriculum changed caused was an overlap in topics and concepts taught in this course for CPE and CSC majors. This is an ongoing problem, but the form of the problem has mutated. The issue is that CPE students are required to take CPE 315, an architecture course offered by the CSC department. Many professors in the CPE program have complained that the overlap is bad for CPE students. The truth is that CSC department chose to no longer require their students to take CPE 133 and CPE 233. As a result, CSC students have little or no experience with actual digital hardware or hardware design in general. That being the case, it's a mystery to me how you can teach a junior-level computer architecture course (CPE 315) without having a clue about basic digital hardware. The current push is to target CPE 233 as the problem. The reality is that CPE 233 exists in large part to support EE students and we've been able to fight off the notion of changing CPE 233 in order to support the shortcomings of another department's curriculum.

[1] By current version, I'm referring to the version first taught by Jeff Gerfen which was centered about the design and implementation of the RAT Microcontroller. Other CPE 233 politcially saavy instructors focused on their academic careers rather than their students typically opt to teach other versions of CPE 233.

[2] I got in a lot of trouble for writing this. It's true; the truth only hurts liars; there are a lot of hurt people in academia.

[3] As reported to me by countless students in this instructor's class. The joke continues due to 1) the politics in the EE Department and CPE program, and 2) the strong resistance to any type of change exhibited by a vocal minority of EE and CPE faculty. The only thing that allowed change to occur in this area was the creation of CPE 233; the studio-version of this course allows instructors to operate independently of each other, thus protecting individuals from the politics.

3. CPE students taking CPE 233 most likely have more programming experience than EE students based on the notion that CPE students take CPE 123-101-102-103, while EE students only take CPE 101. Although this is an issue, programming only comprises about 40% of this course. Moreover, the programming is low-level (assembly language) and is the first time in either the CPE or EE curriculum that students see the material.

1.4 Issues with "Modern Computer Design"

If you ask a hundred people to define the notion of a computer, you will surely receive a hundred different replies. As you know (or as you'll soon find out), if you search for a common and usable definition of a computer, you may only find a description at such a high level, that the definition is almost worthless. Moreover, if you pick up a book on computer design (and there are hundreds of them out there), each book will take a different approach to describing what a computer is and what a computer does (and this of course does not include the different programming languages these computer books describe). There was a time (maybe in the 1950's) where there was a definition of a computer that described actual computers more completely, but the ever-expanding field of computer science and computer technology quickly muddied that definition.

I've spent a significant amount of time trying to figure out how to arrange a book on computer design such that it makes the topic both relatively easy to grasp and somewhat interesting to work with along the way. The main problem is that it is hard to describe something until you know what it does, and you generally won't get a good feel for what it does until you do it, but you can't do it until you know what you're doing, but you don't really learn things until you do them… I'm thinking you get the picture. The final result of this dilemma is what you are reading now. My basic solution to this dilemma is to divide FRCD into four sections.

In a perfect world, where authors write perfect books, each chapter in the book would lead nicely into the next chapter and no chapter would assume knowledge contained on a page after the current page you're reading. I've divided FRCD into four parts: one part is an introduction and basic digital design review, one part contains general knowledge regarding some of the supporting topics regarding computer design; another part of the book contains topics specific to the computer we design in this text, and the final part contains information about assembly language programming concepts. The notion here is that you'll be reading from four different portions of the book as you work towards obtaining a grasp on the basic concepts in this book.

1.5 The RAT Microcontroller/Microcomputer

This textbook describes the development and basic concepts of the RAT Microcontroller (MCU) or RAT Microcomputer. We'll comment on the specifics of a microcontroller in a later chapter. There are many ways to introduce the concept of computer design; this textbook takes the approach of having you design the RAT MCU using VHDL models. You can then synthesize your models such that you can test your own personal version of the RAT MCU on configurable hardware such as an FPGA (more comment on this later).

There is nothing overly special about the RAT MCU, but there is some history associated with it. Knowing this history may help you become more comfortable with the approach this book takes to describing the RAT MCU. Here are the preliminaries:

- The name RAT has no special meaning. I personally like the notion of RATs and such things, as I formerly was involved in a group of musicians that utilized "RAT" in their name. Hey, rats are smart, clever, and simple; no surprise that I still like and connect with the name. If you can pretend RAT is an acronym, I'm all ears for telling people that RAT actually stands for something (many people have seemed troubled when I told them that RAT doesn't stand for some meaningful technical term).

- The RAT MCU started out as a concept for a computer design course. The Cal Poly Electrical Engineering department was suddenly required to teach a computer design/assembly language course in a ten-week period, but there were no materials out there to support such a knowledge-impacted course. The first attempt at such a processor was the ESX MCU, which I designed in the summer of 2004. The ESX MCU was a great learning experience, but it never went anywhere due to some oversights with the design. The main problem with the ESX MCU was that I designed it to be a subset of the Atmel AVR line of MCUs, something that felt like a good idea at the time but turned out to be rather pointless and stupid.

- A really cool student (Kianoosh Salami) and I "designed" the RAT MCU in the summer of 2009. The design simply comprised of a minimal set of instructions (we'll discuss exactly what that means later) that the RAT MCU would support. Our main goal was to make the instruction set as small as possible but result in a meaningful, synthesizable, and useful computer. Note that we never designed any actual hardware: the design started out as simply an instruction set. Our inspiration for the RAT MCU design was partially driven by our experience using the PicoBlaze2 and PicoBlaze3 MCUs in an older version of the course. The PicoBlaze designs represent a working MCU defined with VHDL models. The PicoBlaze3 was an improvement over the PicoBlaze2 design; naturally, the RAT MCU is an improvement over the PicoBlaze3 design[4]

- We define the RAT MCU primarily using VHDL behavior modeling. Recall that with behavioral modeling we need to describe the behavior of a circuit at a relatively high level rather than describing the underlying logic that implements that behavior. The notion here is that if you can successfully describe the operation of the circuit, you must therefore understand the how the circuit works and how the circuit interacts with other circuits in your RAT MCU. In my experience teaching the course, this is for the most part true (woo hoo!). The result is a working MCU, though the footprint is not as small as it could be based on the behavioral modeling approach.

- In the Fall of 2010, I pitched the RAT MCU concept for a course to Jeff Gerfen. He apparently liked the concept enough because he agreed to use it for his CPE 233 course he would teach in the Winter 2011 quarter. At that point, I agreed to support his efforts by generating an assembler for the course. I wrote the assembler for the course in the Fall 2009 quarter based on the instruction set Kianoosh and I had previously designed. I worked with Jeff to refine the assembler and add some features during the Fall 2010 and Winter 2011 quarters. Included with this was the "RAT Assembler Manual", which describes the instruction set and the various features contained in the assembler. In reality, Jeff made the course happen. He basically started from scratch on the course based on an instruction set and the promise of an assembler and assembler manual. He thus did all the major initial development work for the course, which is a significant feat that you cannot overstate[5].

- When the course was first taught in Winter 2011, two students created the RATSim, which serves as a simulator and debugger for the RAT MCU. The RATSim also serves as an editor and calls the assembler when directed to do so. This piece of software is integral to making the course meaningful.

- Other instructors including Bridget Benson, Kari Haworth, Jeff Gerfen, and myself further developed the CPE 233 course. These changes underscore the good things that can be done when

[4] The notion of improvement needs defining here. The PicoBlaze designs were highly optimized to create a fast MCU that synthesized into a small footprint. The PicoBlaze design did not use much VHDL behavioral modeling. The resulting model was great, but the models did not support actually understanding how a computer works.

[5] For those people who don't know what is involved in developing a course, here are some of the issues. In a perfect world, the administration would provide instructors with as much time as the required in order to present a "good" course. In reality, the administration provides no time at all for instructors to develop courses. Instructors are not required to develop courses, but good instructors don't feel comfortable presenting bad courses; a bad course in this instance is one that does not present the course materials in an coherent and modern manner, which is an ongoing problem in fast-moving technologies such as digital electronics.

instructors work together and share their work, something that was unheard of in the days when old-fart wankers ruled the digital area of the EE and CPE departments with an iron-fist; dark days indeed.[6]

- The lack of viable book for CPE 233 has always been a problem. The first text arose from the ashes in Spring 2014, and has been in constant development ever since. As I proof read this, I'm done editing version 9.00 of a course text. It's a work in progress… I just added 50 pages to the text over the summer (2016). It's a mystery why I did not include this information earlier in the game. The text becomes better with everything I add and everything I fix, but there is still much room for improvement.

[6] And the worst part of it all was that many of the asswipes who were trying to control the course had never taught the course and had no intention of ever teaching the course. Why is it that people become corrupt when they find themselves in a position of power?

1.6 Chapter Summary

- **Course History:** This textbook was originally designed to support a new course in the Electrical Engineering Department. This new course started out as CPE 229/CPE 269, but later morphed into CPE 233. This course replaced a course in advanced digital design and assembly language programming. This course and textbook is under constant development.

- **Course Conception:** This course and subsequent support materials was originally conceived of by Kianoosh Salami and Bryan Mealy. The aim of the design was to have a computer with a small instruction set have it be large enough to be both useful, versatile and facilitate the understanding of low-level computer design and basic assembly language programming concepts.

- **Course Progress:** This course is a result of several instructors working together and sharing their work. This sort of collaboration is not typically found in academic environments due to the administration underlying approach of judging instructor's performance on something other than an absolute standard, which results in instructor's having being reluctant to share their work. In this scenario, students always lose.

1.7 Chapter Exercises

1) Briefly explain why is there no great definition of a computer that satisfies everyone who may be asking such a question.

2) Briefly describe why it is hard to define the notion of a computer.

3) Briefly explain why there is no good off-the-shelf textbook for this course material.

4) Briefly describe the four parts of this textbook.

5) Briefly explain why this text divided into four parts.

6) Briefly comment on where the name RAT came from.

7) Briefly describe the main problem with the ESX MCU.

8) Briefly describe how the RAT MCU started out.

9) Briefly describe the main features of RATSim.

2 The Basic Computer in High-Level Terms

2.1 Introduction

The purpose of this chapter is to describe the notion of "computers" at a high level using terms associated with computer programming and basic digital design. CPE 133 was the first course in digital design, and having course experience in computer programming is currently a pre-requisite for CPE 133. This is an important chapter as it gives you a meaningful roadmap for the stuff you'll be learning from this book and the associated laboratories. As you may or may not know, FreeRange Digital Design is my book on digital design; it is available for free download from various websites (any viable search engine can find a version).

Main Chapter Topics

- **HIGH-LEVEL OVERVIEW OF COMPUTER ARCHITECTURE:** This chapter provides a high-level overview of computer architecture, which provides a context for the information in this text.
- **COMPUTER PROGRAMMING CONTEXT:** This chapter provides a context for the act of programming computers in terms of hardware, software, and the human destine to interact with them.
- **LEVELS OF PROGRAMMING:** This chapter describes the various levels possible for programming computer.
- **COMPUTER PROGRAMMING CONTEXT:** This chapter provides a context for the act of programming computers in terms of hardware, software, and the human destine to interact with them.
- **DIGITAL DESIGN HIERARCHY:** This chapter provides a reminder of the path we've taken to arrive at the point of designing a computer.

Why This Chapter is Important

This chapter is important because it provides a high-level overview of the course in terms that you should already know. This overview provides a roadmap of the course and the text.

2.2 What is a Computer?

In truth, if you ask a hundred people what a computer is, you'll most likely receive 100 different replies. The working definition for a computer that we'll go with for now is this: *a computer is a device that sequentially executes a stored program*. This so-called "device" is generally some special set of hardware that someone has configured to operate in a useful way with the stored program. The underlying factor is here is that as a result of executing a program, we'll end up with some useful result. We consider the program to be software;

this software executes on the computer's hardware. This definition of a computer works for now; we'll for sure be adding to it in later chapters.

The only thing that a computer can provide us with is data: 1's and 0's; it's up to the user to interpret this data in such a way as to make the 1's and 0's into actual information. The real purpose of a computer is to process the data according to the directions contained in a stored program and return something useful the form of bits. In the end, a computer may be nothing more than a device that twiddles bits. This view of a computer allows us to model a computer with the standard block diagram shown in Figure 2-1.

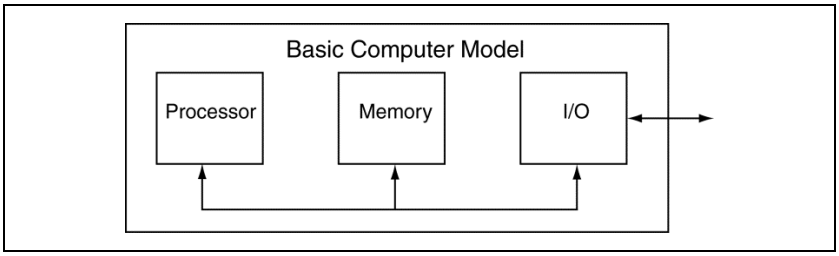


Figure 2-1: General model of a computer.

The three blocks in Figure 2-1 deserve some explanation, as these are common terms in the world of computer design. We'll delve more into these later, but for now, here's a short overview.

- The processor is a generic term for a module that inputs data, does something to it (such as processes it), and delivers the result somewhere. The processor is the "brains" of the computer, which means its main responsibility is to crunch data as required by the program stored in memory and being run by the computer.

- The memory is one of the few words in computer design that is not an acronym. The memory holds data that allows the computer to operate properly. In short, the memory module, at the very least, stores the program the computer is executing. In reality, there are many other pieces of "memory" in a computer; we'll get to those later.

- The I/O is short for Input/Output. For any computer to be useful, it needs to communicate with the outside world. In a generic sort of way, the computer receives input data from the outside world (input such as a keyboard press or sensor data) and then delivers some result back to the outside world (output such as display device or audio device).

Outside of the modules in Figure 2-1, the other important item is the directed arrows. In this overly simplified drawing, the arrows indicate that the CPU connects to the memory and the I/O, which indicates that it is exchanging data with these devices. Likewise, the Memory connects to the CPU and the I/O. Note that only the I/O connects to the outside world.

Figure 2-1 lists a computer model that provides an opportunity to present one of the most commonly used words in the world of computer design. Namely, Figure 2-1provides a description of a computer *architecture*. The word architecture is the commonly accepted method of describing how a computer works at a relatively high level. More specifically, the computer architecture depicts the arrangement and interconnection of a computer's functional blocks. The architecture shown in Figure 2-1 is quite high level but it does provide useful information. One of the problems with the use of the words "computer architecture" is that it is not specific to any one level of description of a computer. This is why when you hear someone use this term, you can never be sure exactly what level of description he or she are referring to.

2.3 You and the Computer

You're either a computer user or a computer programmer or both (and both at the same time). The most basic interaction with a computer is for you to "use" the computer. This roughly means that you're interacting with a physical device that some form of a computer is controlling.

Figure 2-2 shows a model of this simple interaction using a cheesy diagram. In Figure 2-2, "You: the user" is interacting with the computer. This means that somehow you are providing the computer with data. This data may come from typing on a keyboard of some type of sensor data such as a heart monitor. We model this interaction in Figure 2-2 with an arrow directed away from "You: the user" and going into the box label "Computer". Note that the Figure 2-2 model of a computer only show an interior box label "memory", which emphasizes the notion that we consider the computer be "running" because it is executing a set of instructions (a program) stored in its memory.

For the computer to be actually useful, it must return data to you. This data could take on any forms such as a visual display, a blinking LED, a buzzing, etc. The computer generates the data it provides you with and outputs that data by the running program; the program most likely under constant influence by your input as "You: the user". Yes, a simple model indeed. You embody this model about a bajillion times each day, but who's counting?

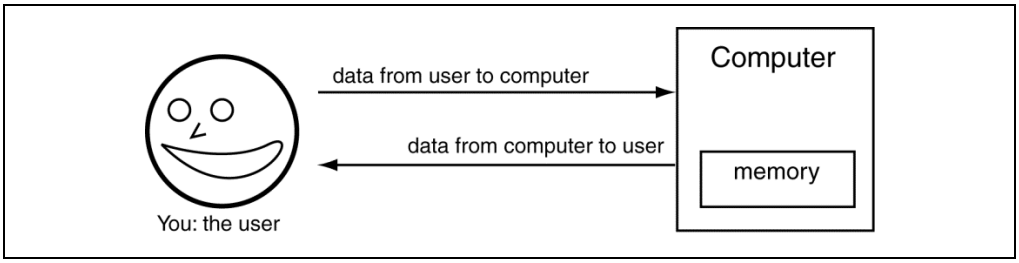


Figure 2-2: A basic model of an eerie human-like face interacting with a computer.

You don't always have to be a simple user of a computer; you can also write computer programs. Figure 2-3 shows a diagram that models you as the programmer (labeled "You: the programmer"). There are several steps for you to program the computer.

- The first step is that you have to write a "computer program". The notion here is that you use some type of "computer language" and some form of software (such as a text editor) to write your computer program. The "computer language" is generally some sort of textual set of instructions for the computer.

- The second step translates the computer program to something that the computer can understand and use. You generally write the program in a language you can understand, and then you input it into another piece of software that translates the instructions in the computer language you're using into a stream of 1's and 0's. The computer itself is a digital device and thus can only understand 1's and 0's.

- The third step is go get the 1's and 0's that make up your program into the memory of the computer. We're going to leave this step outside this conversation due to the notion that there are many ways to do this and we want to keep speaking in generalities.

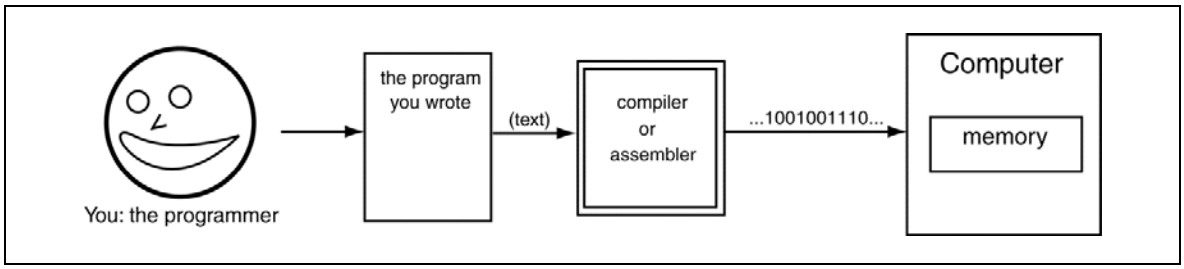


Figure 2-3: A basic model of an eerie human-like face writing a program to be executed a computer.

2.4 Computer Architecture: For the Hardware People

We've agreed that a computer is a piece of hardware that executes a stored program. We went over some level details regarding the operation of a computer in a previous section. In this section, we'll leverage your current knowledge of digital design, which allows us to delve deeper in to the basic computer model of Figure 2-4.

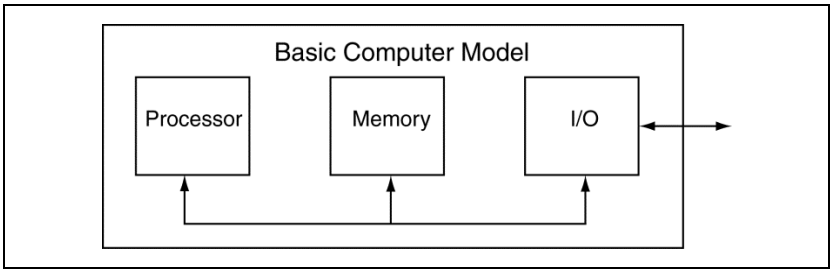


Figure 2-4: About the most basic model of a computer that you'll find.

Figure 2-5 expands on the computer model of Figure 2-4 by listing the useful sub-modules for some of the computer's basic blocks. The goal of this new computer architecture is to provide you with more high-level insight into how a basic computer operates. We'll do this by describing the basic block in more detail.

The I/O module: This module did not change from the previous computer model. One interesting item to note is that the model in Figure 2-5 has the I/O module only connected to the processor. This is arbitrary; different architectures would have different interconnects but this model is an attempt to keep things generic.

The Memory module: A typical digital circuit can contain many types of memory ranging from flip-flops to large structured memory devices. The memory module in Figure 2-5 contains two -: instruction memory and "data" memory. As we spoke of earlier, the instruction memory stores the program that the computer is executing. The data memory stores "data" that the computer requires to obtain its desired result. We use the term "data" memory to mean many things, all of which are outside of the context of this discussion. In short, computers generally store data in various places as a means to obtaining the required result.

The Processor module: We've divided the processor module into two separate sub-modules: the CPU and the Control Unit.

- The control unit is responsible for reading the instruction it needs to execute and sending out the appropriate control signals to the other hardware modules in the computer responsible for executing that particular instruction. Note that the arrow points from

instruction memory to the Control Unit for the instruction and from the Control Unit to the CPU with control signals. The Control Unit is nothing more than a Finite State Machine (FSM), no different from the ones we studied a beginning digital design course. As you'll see later, the Control Unit is responsible for making sure the right things happen at the right time to implement the computer's instructions (which we typically refer to as sequencing).

- The acronym CPU stands for Central Processing Unit. The notion here is that the CPU "processes" data under control of the Control Unit, which is in turn following orders from the instruction in memory. The CPU acts under direction of the control unit. As with many FSMs, the Control Unit receives status of various operations from the CPU as Figure 2-5 indicates with an arrow directed from the CPU to the Control Unit. One of the main sub-modules of the CPU is the ALU, which is an acronym that stands for Arithmetic Logic Unit. The notion of the ALU is somewhat antiquated in that a typical ALU does more than simple arithmetic and logic instructions. The term CPU is also antiquated; in days gone by, hardware was expensive and there was typically only one piece of hardware that did all the number crunching/bit manipulation; the CPU has a "central" location in the hardware.

- The Control Unit is an FSM so it is thus a sequential circuit. The memory associated with an FSM is the state variable storages. The CPU is typically a combinatorial circuit.

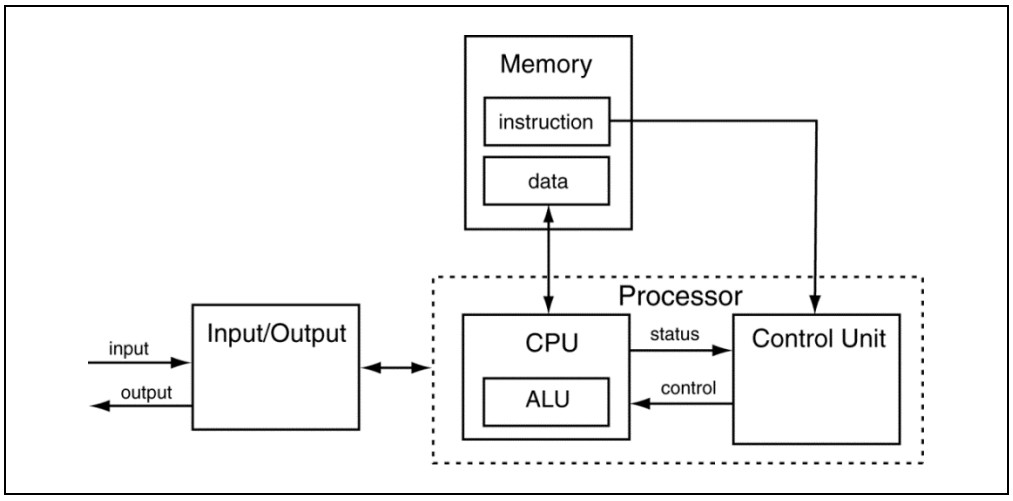


Figure 2-5: A more detailed computer basic computer architecture.

2.5 Computer Instructions

So what exactly is an instruction? As with everything else in a computer, it is nothing more than a set of bits. These bits act as control signals that implement or allow certain data processing operations to occur on a computer. Although it is possible for us humans to write strings of 1's and 0's and use them to represent directives to the some computer hardware, it's not the easiest approach to programming a computer. We refer to programs in the form of 1's and 0's as *machine code* or *machine language*. As you probably can imagine, writing programs in this manner is tedious but possible. Trying to understand a program written in machine code is nearly impossible: you can do it but it only proves you too much time on your hands.

The official handling of the 1's and 0's that make up the instructions is grunt work. In the olden days, people actually did this because there was no better option at the time. A better approach, however, is to use *mnemonics* to describe the set of 1's and 0's that tell the computer how to act. The mnemonic generally describes in short-hand notation the operation the computer should perform. Each of these mnemonics (and

some other associated information) is associated with some specific set of 1's and 0's. The set of mnemonics for a given computer is generally what we consider the *instruction set* for that computer.

When programs are written using instruction mnemonics, we say that the program is in *assembly code*. This assembly code must be converted from mnemonic form to 1's and 0's that the computer can understand (machine code) before being placed in the system memory and used to control the processor. We refer to this translation process as "assembly"; we refer to the piece of software that performs the assembly as an *assembler*. Once again, we refer to the entire set of mnemonics representing the *instruction set* of a computer as an *assembly language*.

The bad news is that there are many different assembly languages out there. There truly needs to be because each different computer needs its own specialized set of instructions in order to do the proper tweaking of the internal hardware. The good news is that on an assembly language level, you're never really doing anything more complicated than twiddling bits. The reality is that the flavors of bit-twiddling are limited (in other words, you can only do so many things with bits). These means that once you know one assembly language (and have a grasp of generic programming concepts), you can relatively quickly and easily switch to another by simply learning the syntax of the new assembly language. Every instruction set has an instruction that rotates a value in a register: in one assembly language the accompanying mnemonic may be ROR and in another language the same function would be RR. Same function, different mnemonic.

2.6 Computer Architecture: For the Programmer People

You're either a user of a computer or a programmer of a computer (you can be both, but generally not at the same time). If you're a computer programmer, you'll need to understand the computer architecture you're programming as well as the tools you have for programming. These two items fall under the notion of the "programmer's model (sometimes referred to as the programming model) and the instruction set. This section describes these two items in terms that you should be familiar with from you previous digital design and computer programming experience.

2.6.1 Programmer's Model

The Programmer's Model is a high-level view of the hardware resources that the programmer can use to write their programs. Note that while a computer is comprises of a relatively significant amount of hardware, the programmer cannot control all of that hardware. Also, note the programmer controls the hardware by writing "instructions", which we'll discuss in the next section.

Figure 2-6 shows the programmer's model for the RAT MCU. Relatively speaking, there is not much there, particularly in comparison to the basic computer hardware of Figure 2-5. Once again, computers may have a lot of hardware, but not all of that hardware is directly controllable by the instruction set; only the hardware in the programmer's model is controllable the programmer by issuing instructions. Here is a brief description of the hardware items in Figure 2-6; this description will show you show you that you already have an understanding of the individual components in the programmer's model; we'll fill in the details of how these items work and how they the instruction set affects them later.

> Register File: The register file is a standard word for a bunch of registers. These are registers, and are thus memory; the RAT MCU uses registers to store intermediate results and supply data to other ALU sub-modules. The RAT MCU uses registers as operands for various types of operations. Figure 2-6 uses the "r0 – r31" notation as a way to indicate there are 32 registers in the RAT MCU.
>
> Program Counter: The program counter, or "PC", is also a type of register (recall that a counter is a special type of register). We consider the PC to hold an address of some instruction in program memory, which is why we often refer to it as "address register". The PC typically sequentially steps

through program memory (an increment operation) of jumps somewhere else in program memory (a parallel load operation). The PC is truly a simple device.

Stack Pointer: The stack pointer, or "SP" as we sometimes call it, is another register, but it is a register with "features". There are many ways to implement the SP, but typically, the SP is a register that has increment, decrement, and parallel loading features (yet it is still a register).

C & Z Condition Flags: Although the complete description is beyond this discussion, the C and Z condition flags are simply flip-flops, or 1-bit storage elements. These flip-flops are generally set and cleared based on the results of operations in the CPU.

Scratch RAM: The scratch ram is another piece of memory. RAM is an acronym for "random access memory". For now, all you need to know is that the RAM stores bits. The RAT MCU uses the RATM to store data (constants, intermediate results, etc.); this data can come from one of several different sources.

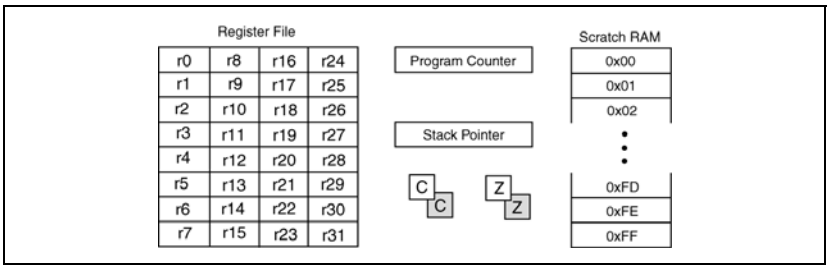


Figure 2-6: The Programmers Model for the RAT microcontroller.

2.6.2 Instruction Set

While the programmer's model shows the resources available to the programmer, the instruction set allows the programmer to use those resources. In other words, the instruction set is what the programmer uses to create an actual program. Figure 2-7 shows the instruction set for the RAT MCU. There are a few important things to note from this instruction set

- Figure 2-7 shows the RAT MCU's instructions in "abbreviated mnemonic" form, which happens to be the basis of assembly language programming. I call this abbreviated because many of the mnemonics have included operands, which I left out because they're not important for this discussion. The notion Figure 2-7 is trying to show is that there are a finite number of instructions that a programmer can execute on the RAT MCU. I'm not sure if 47 seems like a lot of instructions or not many instructions to you, but you can do just about anything with this number of instructions while still maintaining an air of simplicity.

- Each particular computer that you're intending on programming has a finite number of instructions that hardware can execute under program control. Figure 2-7 shows the instruction set for the RAT MCU. Any other computer will have a different instruction set. There is no magic with the instruction set: someone sat down and decided which instructions the RAT MCU would support. There are many trade-offs along the way as far as instruction selection goes, which we'll discuss later.

- Some RAT instructions have more than one format, which is why we list some instructions twice in Figure 2-7. As you find out later, the same instruction can be executed using different sets of operands, such as with the MOV and ADD instructions.

- The good part and bad part about assembly language is that the instructions in Figure 2-7 are all you get. This is good because with so few instructions, there is not much to learn. This is bad because with so few instructions, you'll sometimes have to be creative in how you implement you programs because that instruction you really need (and it was included in the previous MCU you were working with) is simply not in the RAT instruction set.

Program Control			Interrupt	I/O
BREQ	BRN	CLC	RETID	IN
BRNE		SEC	RETIE	OUT
BRCS	CALL		SEI	
BRCC	RET	WSP	CLI	

Logical		Arithmetic		Sh & Rot	Storage	
AND	AND	ADD	ADD	LSL	ST	PUSH
OR	OR	ADDC	ADD	LSR	ST	POP
EXOR	EXOR	SUB	SUB	ROL	LD	
TEST	TEST	SUBC	SUBC	ROR	LD	MOV
		CMP	CMP	ASR		MOV

Figure 2-7: A quick listing of all of the RAT instructions.

2.7 Programming Language Levels

If you're reading this sentence, you've probably programmed a computer. If you've programmed a computer, you hopefully have some notion of the low-level details of what you were actually doing as you programmed that computer. In case you did not know what you were doing, this section aims to give you a quick overview of the big picture.

Once again, the bit patterns that are associated with the instructions control the operation of a computer. You can write a computer program at one of three different "levels"; these levels are: 1) "machine" code, 2) "assembly" code, 3) and some "higher-level" language. This section describes these levels including the information of Figure 2-8.

2.7.1 Machine Code

This is the lowest level of programming. A program written in machine code is nothing more than a set of 1's and 0's which are arranged in bit-patterns that direct the operations that the underlying architecture should perform. The good part about writing programs using machine code is that there is no need to use other software (not including a text editor) as a precursor to writing a program. The downside of this approach is that programs are completely unreadable. There probably was a day when all programmers had to use machine code to write all their programs, but that was back when dinosaurs were responsible for programming computers. Although every program that anyone ever writes eventually ends up as machine code, the programs never start that way. Unfortunately, some instructors still require that students be able to code programs using machine code, which is nothing more than an indicator of the age of the instructor.

2.7.2 Assembly Language

The next level up in the programming hierarchy from machine code is assembly language programming. In an assembly language, we replace the bit-patterns that form the instructions by mnemonics that loosely indicate the purpose of the instruction. The upside of using assembly language programming over machine code is that mnemonics bring a level of understandability to the code as opposed to attempting to use your

human brain to interpret endless strings of 1's and 0's. The downside, (if it we consider it one) is that you need another piece of software referred to as an "*assembler*" to translate the assembly language instructions to machine code. The downside of assembly language programming is that every different computer architecture (the computer hardware) will necessarily have a different assembly language. Although writing code in different assembly languages is not that complicated once you know one assembly language, it does, however, have a slight learning curve. In the end, it's still all about using the assembly language instructions to crunch bits.

2.7.3 Higher Level Languages

The next step beyond assembly language programming is to use some type of higher-level language (HLL). Because each assembly language instruction generally performs only a basic operation, assembly language programs can quickly become long (many lines of assembly instructions) when the program requires a relatively complex set of operations. One possible solution to producing long programs is switching to coding the programs using a HLL. When you use a HLL, each line of code in the HLL can represent many lines of assembly code, which leads to shorter and arguably more understandable programs. When you use a HLL, you must use a *compiler* to translate the HLL code into machine code. Most likely, the compiler first converts the HLL code to assembly code before the final translation to machine code.

Using a HLL has one distinct advantage over assembly code: once you know one HLL, you can write code for any architecture without know anything about the underlying assembly language, assuming you have the correct compiler. This effectively lessens the learning curve for switching processors and generally makes your HLL code independent of the computer architecture your programming. One major downside of HLLs is that the code is not necessarily as efficient as it would be if a human generated the assembly code.

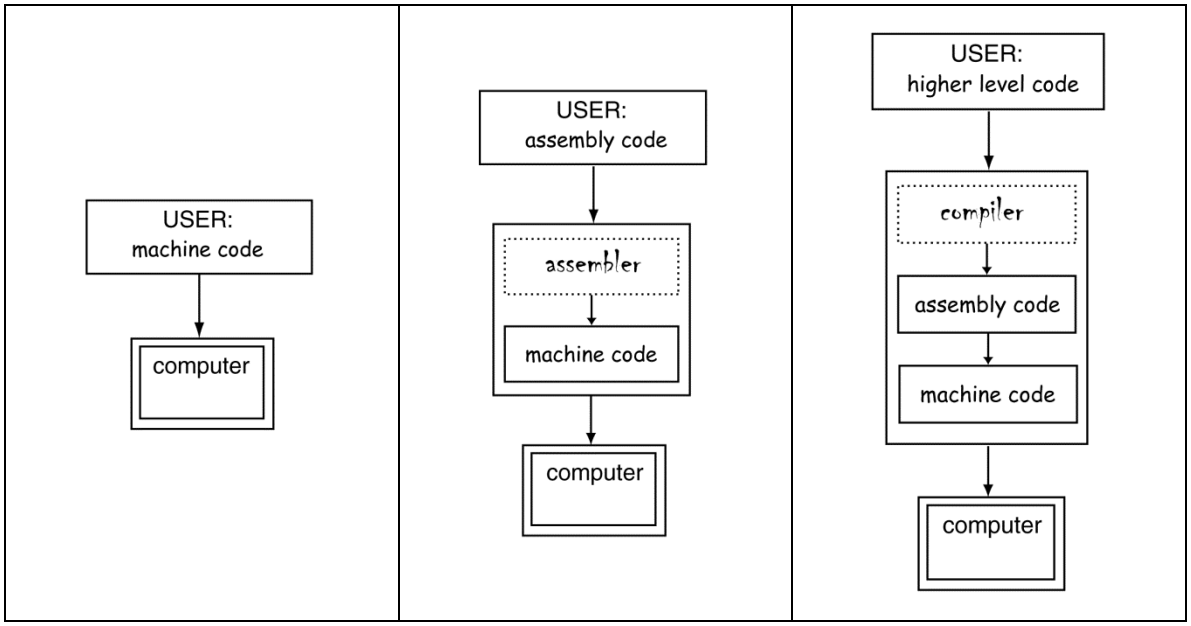


Figure 2-8: The three different levels in which you can program a computer.

2.8 The Digital Design Hierarchy

This text is about moving towards designing a computer. You've come a long way down the digital path to get to this point and here is a reminder of some of the more important milestones along the way. It all started

with your first digital design course. Computer design represents what the next step in the natural progression of your "digital education". Here is a brief reminder of the progression:

- The typical digital design course starts with number systems including a strong emphasis on binary number systems and various methods to represent information in binary form (binary coded decimal, 2's complement, signed and unsigned numbers, etc.). Although this was not digital design, we'll be using many of these concepts directly because we have a sincere interest in the ways computers store and interpret bit patterns.

- Next came the basics of digital design: AND, OR, NOT functions and gates. This quickly got into the design of basic combinatorial circuits with much emphasis on reducing Boolean equations before circuit implementation. The circuits that we implemented at this point were generally pointless but they provided an enjoyable academic exercise.

- Next, we placed the basic gates in certain configurations in order to obtain certain functions. Once we did this, we quickly abstracted the designs to a higher level in order to avoid talking about nasty low-level things such as gates. The resulting devices were more complex than gates but the complexity was manageable because you understood the functionality of the circuit from a high level. These more complex devices included such things as MUXes, decoders, adders, comparators, parity checkers, etc. If you're anything like me, you've long forgotten the exact configuration of gates and things in these devices, but you completely understand how the circuit functions. For example, the mention of the word "MUX" brings to my mind a form of data selection. It does not, however, bring to mind a circuit any more complex than a block box.

- The concept of memory arrived with the introduction of sequential circuits. We used these memory devices (primarily flip-flops) to construct finite state machines (FSMs). The FSM had two primary functions: register-type circuits (counters) and control units (used to control other circuits). The registers represented an abstraction of the basic flip-flop into more complex but highly useful devices such as multifunction registers such as counters, and shift registers. Another abstraction of the basic memory element were FSMs which were advertised as being control circuits to control yet to be named digital circuits.

This progression represented part of the big picture: these were all tools, which we can use to design and understand a basic computer. Note that many times along this progression, we constructed circuits out of small boxes, placed the small boxes into another box, and gave it a new name. This embodies the general approach of computer design and the understanding of complex problems of any type: the *hierarchical* approach. We've applied this approach from early on in your beginning digital design course in that we studied gates rather than the underlying transistors that implemented them. We extended this approach with slightly more complicated circuits which were a special assembly of gates (decoders, MUXes, etc.). And in the end, it was this hierarchical approach that allowed us to understand complex circuits by abstracting upwards. For example, the concept of a 4:16 standard decoder with a chip enable is easy to comprehend while the transistor-level circuit that implements this functionality would fill a page and would not be at all pleasurable to look at.

This hierarchical concept becomes even more important as we move into computer design. We consider a computer to be nothing more than a very complex FSM. The problem is that even the simplest computer has so many possible states that the techniques we've used to design and analyze state machines are essentially worthless if we try to apply them directly to computer design. Because of this, we necessarily need to take a different approach to designing sequential circuits such as computers. The approach we'll be taking in CPE 233 is a hierarchical approach. In particular, it's a top-down approach to the design of computers, which entails describing at a high-level the functional blocks in a computer. We necessarily use a bottom-up approach to model of some of the more important functional blocks.

2.9 Chapter Summary

- A computer is a device that sequentially executes a stored program. Note that this is one of many definitions for a computer; this definition is quite high-level.
- A computer is comprised of three main subsections: the memory, the input/output, and the processor. The processor crunches data based on instructions stored in memory; the input/output allows the computer to interact with the outside world.
- You as a human interact with computer as either a user or a programmer, or both. You the programmer write programs using a text editor; some other piece of software translates your program into machine code or machine language, which is the only language a computer can actually understand.
- The programming side of a computer is defined at a relatively high level using the Programmer's Model and the Instruction Set. The Programmer's Model shows the resources that the programmer can control using the computer's Instruction Set.
- Programmers can program computers at three different levels: 1) machine code, 2) assembly code, and 3) a higher-level code. The application you're working on and your immediate supervisor generally dictate what level you'll be programming at.
- The notion of "digital design" could mean many things. The idea is that you can perform digital design at one of many different levels. As you progress towards more complicated digital designs, and particularly computer designs, you'll be designing at higher levels of abstraction. Though it would be possible to design an entire computer at the transistor level, no one actually does it as design computer at the transistor level because designing at higher levels of abstraction is much more feasible and cost effective.

2.10 Chapter Exercises

1) In your own words, define the word “computer”.

2) List and briefly describe the function of the three main modules of a computer.

3) Briefly describe what is meant by the term “computer architecture”.

4) Briefly describe the main use of a computer architecture.

5) Briefly describe the two sub-modules that form the processor.

6) Would you expect an “ALU” to do more than just arithmetic and logic instructions? Briefly justify and support your answer.

7) Briefly describe what is meant by them “computer instruction”.

8) Why are there so many different assembly languages out there.

9) Briefly describe why computer instructions are represented using mnemonics.

10) Briefly describe what makes it relatively easy to learn a new assembly language once you know one of them.

11) Briefly describe the distinct advantage does using a higher-level language have over using an assembly language?

12) Briefly describe the three levels of programming.

13) How many general purpose registers does the RAT MCU have?

14) How many scratch RAM memory locations does the RAT MCU have?

15) Briefly describe what is meant by the notion of a “digital design hierarchy”.

3 Basic Digital Design Review

3.1 Introduction

The first course in digital design typically entails learning a standard set of combinatorial and sequential circuits. Despite the fact that this set of circuits is relatively small, you can design any possible digital circuit. We keep referring to these basic circuits as being in our "digital bag of tricks", which means we know what these do, how they do it, and easily use them as the building blocks to any digital circuit. Moreover, we generally understand these basic digital building blocks at a high-level; we know how they operate at a low-level so we avoid designing at that low level and opt to design at the modular level.

VHDL provides a viable approach to modeling digital circuits. In addition, we can easily synthesize those VHDL models into working circuits given that we have the correct software/CAD tools. VHDL also provides a method to simulate digital circuits as simulation software can use the VHDL models to generate timing diagrams associated with the functionality of the models.

This chapter provides a quick overview of digital design including combinatorial circuits, sequential circuits, and Finite State Machine design. In addition, this circuit also provides an overview of some of the more important concepts associated with modeling circuits using VHDL. For a complete overview of these topics, please consult an appropriate digital design text such as FreeRange Digital Design.

Main Chapter Topics

- **OVERVIEW OF IMPORTANT DIGITAL VERNACULAR:** This chapter lists and defines some of the more important terms and concepts related to a beginning digital design course.
- **COMBINATORIAL CIRCUIT REVIEW:** This chapter describes the basic combinatorial circuits that everyone should be familiar with including basic gates, half adders, full adders, ripple carry adders, multiplexors, decoders, comparators and parity generators.
- **SEQUENTIAL CIRCUIT OVERVIEW:** This chapter describes the basic sequential circuits everyone should be familiar with including flip-flops and all the major aspects of Finite State Machines (FSMs).
- **APPROACHES TO DIGITAL DESIGN:** This chapter describes the three basic approaches to digital design: 1) brute force design, 2) iterative modular design, and 3) modular design.
- **VHDL OVERVIEW:** This chapter provides an overview of the main points of VHDL and its use in modeling digital circuits.
- **COMPREHENSIVE DESIGN EXAMPLE:** This chapter provides a comprehensive computer-type design example as a basic introduction to designing digital circuits that we can model as "computers".

Why This Chapter is Important

This chapter is important because it describes most of the important concepts from a typical beginning digital design course. In particular, this chapter provides a fast overview of the topics presented in FreeRange Digital Design.

3.2 Important Digital Vocabulary

If you only remember a few things from CPE 133, you should remember these things. They probably won't help you pass CPE 233 or other courses, but they may help you pass in interview because even a substandard HR person can gauge whether you know these items or whether you're describing a sack of dead chi. As for vocabulary, there are 25 pages of vocabulary in the glossary of the FreeRange Digital Design textbook. Consider browsing that stuff if none of this is making sense.

Functionally Complete: This refers to the fact that some logic gates have the ability to implement all basic logic functions while others do not. NAND & NOR gates are functionally complete for example, because a NAND gate can be used to implement an AND, OR, or an inversion function. This is not true for AND & OR gates so they are not considered functionally complete.

Combinatorial vs. Sequential Circuits: The rough explanation is that sequential circuits contain memory while combinatorial circuits do not. In other words, a sequential circuit has the ability to "remember" at least one bit while combinatorial circuits do not. The better and longer explanation is that outputs of combinatorial circuits are a strict function of the circuit's inputs while in a sequential circuit, the outputs are a function of the sequence of the circuit's inputs. We derive the notion of a finite state machine (FSM) from this previous definition.

Mealy vs. Moore FSMs: In a Moore-type FSM, the circuit outputs are only a function of the state variables. In a Mealy-type FSM, the outputs are a function of both the state variables and the external inputs to the circuit.

Set-up and Hold-times: Generally speaking, in edge-sensitive devices, the non-clocking inputs to a device must be stable (non-changing) for a given period of time both before and after the active clock edge. The setup time refers to the time the inputs must be stable before the active clock edge while the hold-time refers to the time the inputs must be stable after the active clock edge. If you violate setup and/or hold times, the circuit will probably not work because the circuit will be "metastable". Metastability generally refers to the characteristic of the devices output as being neither high nor low and… stuck in the netherworld.

Latches vs. Flip-flops: Both latches and flip-flops are 1-bit storage elements. The difference is that flip-flops are latches that are "edge sensitive" meaning that the flip-flops outputs can only change on an active clock edge. We consider the latch to be a level-sensitive device, meaning roughly that the outputs can change anytime.

In CPE 133, the first topic at hand was to learn the basics of digital design. This included the basic logic functions such as AND & OR, but was more specifically designed towards the gates that implemented these functions. The circuits you initially designed were primarily gate-level, which you abstracted up from the transistor level. The next part of the course used those logic gates to build what we consider standard digital circuits such as multiplexors, decoders, adders, parity generators. These are all considered combinatorial circuits. The next part of the course introduced memory sequential circuits with the introduction of memory elements such as latches and flip-flops. The main use of sequential circuits in CPE 133 was as state registers in FSMs.

CPE 133 initially modeled circuits at a gate-level. As the circuits became more complex, gate-level modeling became more cumbersome which initiated a switch to modeling digital circuits using block-level design. In the lab portion of CPE 133, you implemented digital circuits in actual hardware. Specifically, we modeled the circuits using VHDL and implemented on the FPGA-based development board.

Table 3.1 represents an attempt to use the term modeling in a context that you're somewhat used to hearing. This table provides an overview to the circuits you used in CPE 133 as well as a reference as to how you implemented them. As you can see, you'll not be busying yourself with nasty things like little wires in the CPE 233: we modeled all of the designs in VHDL and implemented them on the FPGA development board. The main portion of the course involves implementing the RAT MCU; the final portion of the course involves primarily involves assembly language programming of the RAT microcontroller you implement. The microcontroller is 100% implemented on the development board.

Course	Design Focus	Circuit Models	Circuit Implementations
CPE 133	basic logic: gates, circuit minimization	gate-level	SSI (small scale integration) IC's
	combinatorial circuits: decoders, MUXes, adders, parity generators	gate-level, VHDL	SSI, FPGA
	sequential circuits: latches, flip-flops, FSMs	VHDL	FPGA
CPE 233	FSMs, counters, registers	VHDL	FPGA
	computer architecture	VHDL, block diagrams, RTL level	FPGA
	assembly language programming, RAT Microcontroller	RTL level, (programmers model)	FPGA

Table 3.1: Models and circuit implementation for CPE 133 and CPE 233.

The main theme of CPE 133 was digital design. In that we all aspire to be great digital designers, we want to be able to generate digital designs as efficiently as possible. While we could implement all of our designs at the gate-level, this would not be efficient. A better approach would be to implement designs at the "block level" or "object-level", or what the previous textbook referred to as "modular design". The general theme of this design approach is to use "black-box" models of known circuit elements (such those listed in Table 3.1 or Figure 3-1) to model digital circuits are a relatively high level. This type of design is extremely efficient because so nicely supports two important concepts in digital design-land: 1) the concept of hierarchical design, and, 2) the ease at VHDL can implement block-level designs.

The presence of large design libraries full of digital devices waiting to be used by a crafty digital designer fully supports the notion of modular design in VHDL. VHDL is a tool that allows you easy access to these device libraries. As you will soon find out, the reality in digital design land is that there are only a relatively few number of core digital devices out there. Even then, you can decompose even the most complex digital circuit into a set of these core digital devices. This decomposition is a reversing of the hierarchical design process. If you are able to understand the operation of the core digital devices, you'll also be able to understand any digital device, regardless of its complexity, if you have the inclination to actually dig that deep into the details.

Figure 3-1 shows a quick overview of digital design as it relates to introductory digital design. What you'll hopefully see from Figure 3-1 is that there aren't that many standard digital devices (or modules) out there and the ones that are out there, are generally very simple devices. Digital circuits become complicated only after you toss down a bunch of these modules into a design. Hierarchical design mitigates this complexity.

In summary, here's all I know about digital design:

1) Digital design is based on a relatively small set of digital devices.

2) Digital design relies heavily on various modeling approaches, particularly object-level design.

3) Digital design modeling relies heavily on hierarchical modeling.

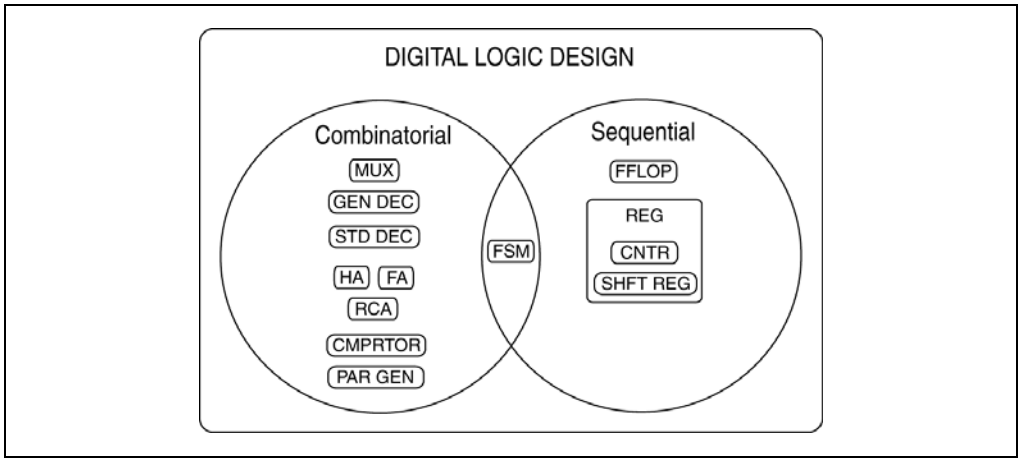


Figure 3-1: The quick digital design overview (most of which was covered in CPE 133).

All digital circuits fall into one of two categories: combinatorial or sequential. In combinatorial circuit, the outputs are a direct function of the inputs whereas in sequential circuit, the outputs are a function of the sequence of inputs to the circuit. This is of course a fancy way of stating that sequential circuits have the ability to "memorize" values while combinatorial circuits do not. Lastly, the ability to "remember" bits in a circuit comes from the fact that somewhere in a sequential circuit is a feedback path from the outputs to the inputs. Thus, combinatorial circuits do not have such feedback paths.

3.3 Combinatorial Circuits

The following is the list and brief description of combinatorial circuits that you should know about based on the assumption you were coherent in CPE 133. Please consult the appropriate text for full explanations of these circuits.

3.3.1 Basic Gates

Though there are only a few gates out there in digital land, there are enough gates to make you forget about some of the details. In addition, once you become a more accomplished digital designer (meaning that you do most of your design at higher levels of abstraction), you tend to forget about low-level stuff such as gates. Keep in mind that an inverter is not really a gate so we do not list it here. Figure 3-2 shows the list of basic gate types associated with AND & OR gates. Recall that AND & OR gates have at least two inputs but can have as many as the designer feels necessarily. The definition of XOR & XNOR gates states that these gates only have two inputs.

- **AND Gates**: two forms: AND form and OR form;
- **OR Gates**: two forms: OR form and AND form
- **NAND Gates**: two forms: AND form and OR form

- **NOR Gates**: two forms: OR form and NOR form
- **XOR Gates**: one form
- **XNOR Gates**: one form (aka: equivalence gates)

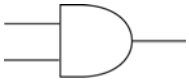
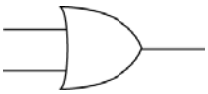
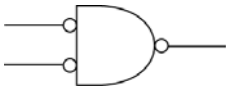
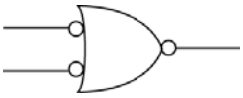
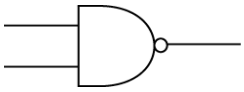
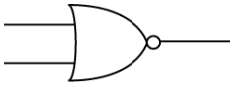
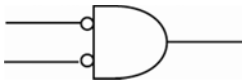
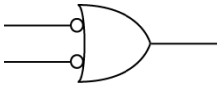
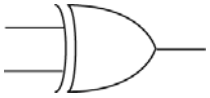
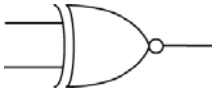
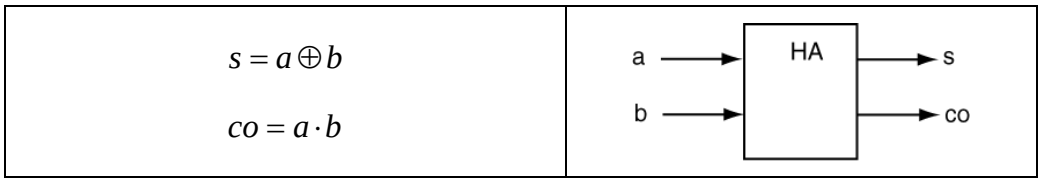
Standard Gates	
	
AND form of **AND** gate	**OR** form of **OR** gate
	
OR form of **AND** gate	**AND** form of **OR** gate
	
AND form of **NAND** gate	**OR** form of **NOR** gate
	
AND form of **NOR** gate	**OR** form of **NAND** gate
	
XOR gate	**XNOR** gate

Figure 3-2: The giant summary of logic gates.

3.3.2 Half Adder

The Half Adder (HA) is generally the first circuit that all digital design students design. The HA is a one-bit adder (adds two one-bit values) and outputs a one-bit sum and a carry-out. The HA is generally designed using a truth table (brute force design or iterative design). Figure 3-3(a) shows the equations describing the HA's two outputs while Figure 3-3(b) show the associated dark box model. The HA is somewhat useful for all those occasions where you may want to add two one-bit values together.

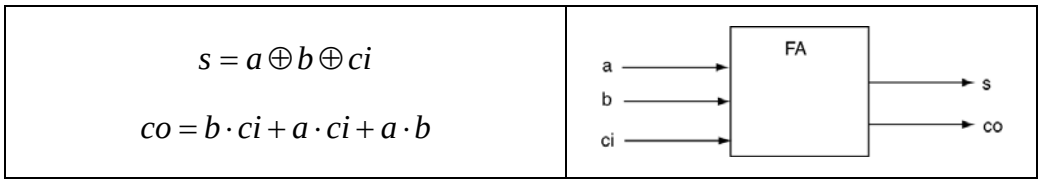


(a) (b)

Figure 3-3: Boolean equations describing sum and carry-out outputs of the HA and a block box diagram.

3.3.3 Full Adder

Once you figure out that a HA is not too useful, you move onto designing a Full Adder (FA). The FA is almost the same as the HA but the FA has an extra input which is generally considered the carry-in (meaning it's the carry in from a carry-out output of some other FA or HA). Figure 3-4(a) shows the equations describing the FA while Figure 3-4(a) shows the associated dark box model.

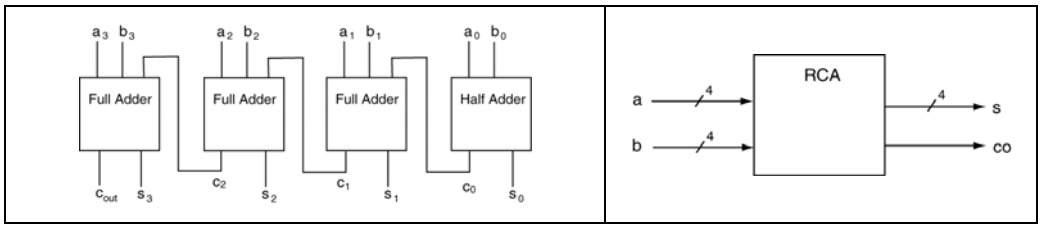


(a) (b)

Figure 3-4: (a) Boolean equations describing sum and carry-out outputs of the FA, (b) the associated dark box model (DBM).

3.3.4 Ripple Carry Adder

Once you realize that there is not too much opportunity out there for Half and Full adders, you generally move onto the ripple carry adder (RCA). The RCA is generally the first circuit you design using iterative modular design (IMD) noting that the 4-bit adder shown in Figure 3-5(a) would have required a truth table with 256 rows had been designed using iterative design techniques. The IMD technique easily extends the 1-bit adding elements (HAs and FAs) to create multi-bit adders. Note that often times the HA in the LSB position of the adder can be substituted with a FA which gives the ability to make larger RCAs (wider, or more bits) by connecting the RCAs in a cascade formation. Figure 3-5 (b) shows the black box model of a RCA while Figure 3-5(a) shows the RCA one level below Figure 3-5(b). Note that the RCA shown in Figure 3-5(a) can include a carry-in input if the HA in the LSB position can be replaced with a FA.



(a) (b)

Figure 3-5: The guts of a RCA (a), and the black box model of a RCA (b).

3.3.5 Multiplexor

The multiplexor, or MUX, is probably the most useful digital circuit element and the most misunderstood. The MUX is an element that "selects" or "chooses" between two or more elements. The inputs to a MUX are the things "being chosen" and control lines to do the actual choosing. The control lines have the typical binary relationship to the circuit inputs in that one control line can choose between two (2^1) items to appear on the MUX outputs, two control lines can choose between four (2^2) items to appear on the circuit outputs and so on. Generally speaking, MUXes only have one output but you can design a "MUX" with more than one output (the general thought is that your circuit is considered a MUX if it's choosing some input or inputs to appear on the output). The "things" that a MUX can choose between include single signals or bundles. Figure 3-6 shows the inner workings of a 4:1 MUX; this circuitry is something you should immediately recognize.

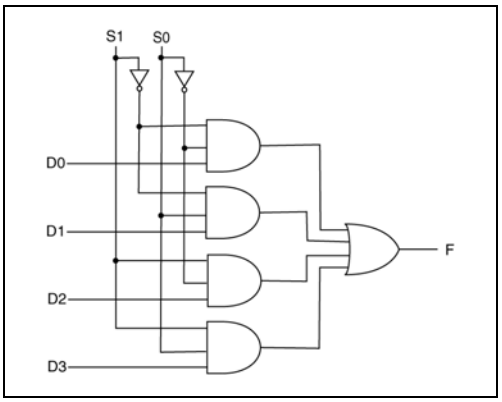
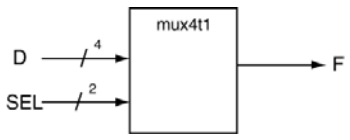


Figure 3-6: The well-known guts of a basic 4:1 MUX.

Figure 3-7(a) shows the dark box model for a 4:1 MUX while Figure 3-7(b) shows an associated VHDL model. MUXes can be model in VHDL using many different approaches. The model in Figure 3-7(b) happens to use selective signal assignment.




```
entity mux4t1 is
   port (   D : in std_logic_vector(3 downto 0);
          SEL : in std_logic_vector(1 downto 0);
            F : out std_logic);
end mux4t1;

architecture mux4t1c of mux4t1 is
begin
   with SEL select
   MX_OUT <= D(3) when "11",
             D(2) when "10",
             D(1) when "01",
             D(0) when "00",
             '0'  when others;
end mux4t1c;
```

(a) (b)

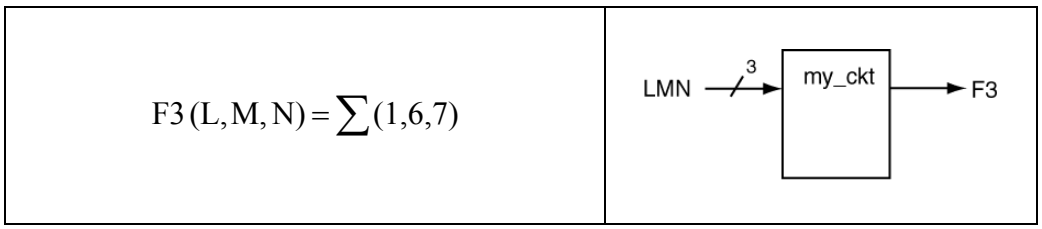
Figure 3-7: The black box diagram and a VHDL model for a 4:1 MUX.

3.3.6 Generic Decoder

The "Generic Decoder" is the name given to any combinatorial circuit that implements a combinatorial circuit that you can't label as some other standard digital circuit. Often time in digital design land, you'll need to

implement a circuit with some "input/output relationship" in order to satisfy your need. Since we're talking about combinatorial circuits here, this means you're generally implementing some functional relationship. Here is the bottom line: any time you need to implement a function, don't bother trying to reduce it using CPE 133 techniques. Instead, try to use some this of generic decoder implementation. Keep in mind that reducing functions is generally not something you often need to worry about when you're modeling circuits with VHDL.

For your pleasure, Figure 3-8 and Figure 3-9 show some random function and its associated dark box model and VHDL model. Anytime I need to implement a function, I use this form (if I can find an example because I don't have this VHDL syntax memorized).



(a) (b)

Figure 3-8: Some random function (a), and its black box model (b).

```
entity my_new_ckt_f3 is
    port ( LMN : in std_logic_vector(2 downto 0);
             F3 : out std_logic);
end my_ckt_f3;


architecture f3_1 of my_new_ckt_f3 is
begin
   with (LMN) select
   F3 <= '1' when "001" | "110" | "111", -- listing the minterms
         '0' when others;
end f3_1;
```

Figure 3-9: A VHDL model for the circuit listed in Figure 3-8.

For yet another example, Figure 3-10 shows a truth table defining several functions while Figure 3-11 shows the associated black box diagram. Figure 3-12 shows the associated VHDL model. Note that this would have been a giant pain in the arse if you were to try to reduce this by hand. Also note that this type of implementation looks a lot like a BCD to 7-segment decoder implementation (hopefully, you've done one of those before).

A	B	C	D	T1	T2	T3
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	0	0	0
0	1	1	1	1	0	1
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

Figure 3-10: A truth table for a set of functions, which can be represented using a generic decoder.

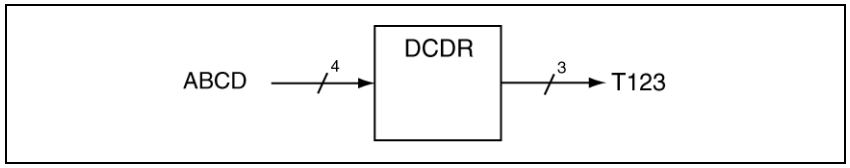


Figure 3-11: Black box diagram supporting the given truth table.

```
entity dcdr is
   port ( ABCD : in std_logic_vector(3 downto 0);
          T123 : out std_logic_vector(2 downto 0));
end dcdr;

architecture dec_dataflow of dcdr is
begin
   with ABCD select
   T123 <= "110" when "0000",
           "000" when "0001",
           "100" when "0010",
           "010" when "0011",
           "000" when "0100",
           "101" when "0101",
           "000" when "0110",
           "101" when "0111",
           "010" when "1000",
           "000" when others;
end dec_dataflow;
```

Figure 3-12: The VHDL model for the generic decoder-type implementation of the given truth table.

3.3.7 Standard Decoder

The Standard Decoder was not use much in CPE 133 due to the general simplicity of the circuits used in that course. There was a time when functions were implemented using standard decoders and MUXes, but the dinosaurs are gone also. We use the standard decoder in many circuits, and we'll visit it later in this text. Different flavors of standard decoders are often labeled as DMUXes, but we'll avoid using such terminology here. Figure 3-13 shows a diagram gate-level diagram of a standard 2:4 decoder. You should recognize this circuit as it represents a major portion of a MUX. There is also a binary relationship between the circuit's inputs (which are select inputs) and the circuit's outputs. If the standard decoder has one input, there are two (2^1) outputs; if the standard decoder has two inputs, there are four (2^2) outputs, and so on.

Figure 3-14(a) shows a black box model of a standard 2:4 decoder while Figure 3-14(b) shows an associated VHDL model. Once again, there are many ways to model standard decoders using VHDL so there is nothing special about this particular model.

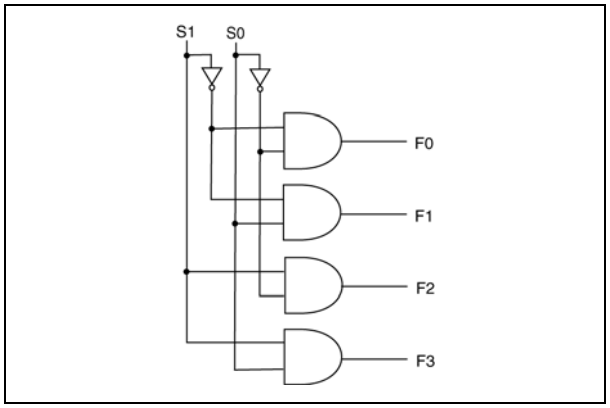
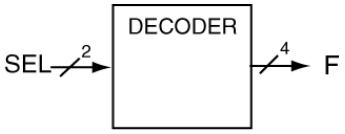


Figure 3-13: The important underlying details of a standard decoder.




```
entity DECODER is
   Port ( SEL : in std_logic_vector(1 downto 0);
             F : out std_logic_vector(3 downto 0));
end DECODER;

architecture my_dec of DECODER is
begin
   with SEL select
   F <= "0001" when "00",
        "0010" when "01",
        "0100" when "10",
        "1000" when "11",
        "0000" when others;
end my_dec;
```

(a) **(b)**

Figure 3-14: A black box model and a VHDL model for a standard 2:4 decoder.

3.3.8 Comparator

The comparator is another common digital circuit. While comparators in general can come in many different forms, Figure 3-15 shows the general form. It is referred to as a general form because there is only one output (indicating whether the two inputs are equal or not). Other less general comparator forms include outputs such as "great than or equal", "greater than", etc. The classic things to remember about comparators are 1) they involve EXOR-type functions, and, 2) we generally design them using iterative modular design (IMD).

Figure 3-15(a) and Figure 3-15(b) show black box models and circuit implementation of a 2-bit comparator, respectively. Figure 3-16 shows that VHDL easily models the comparator with a behavior model.

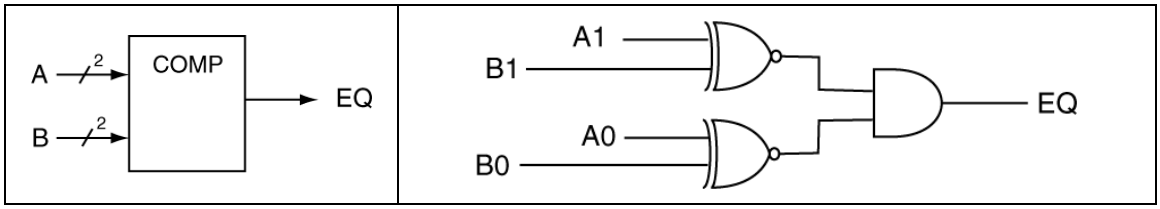


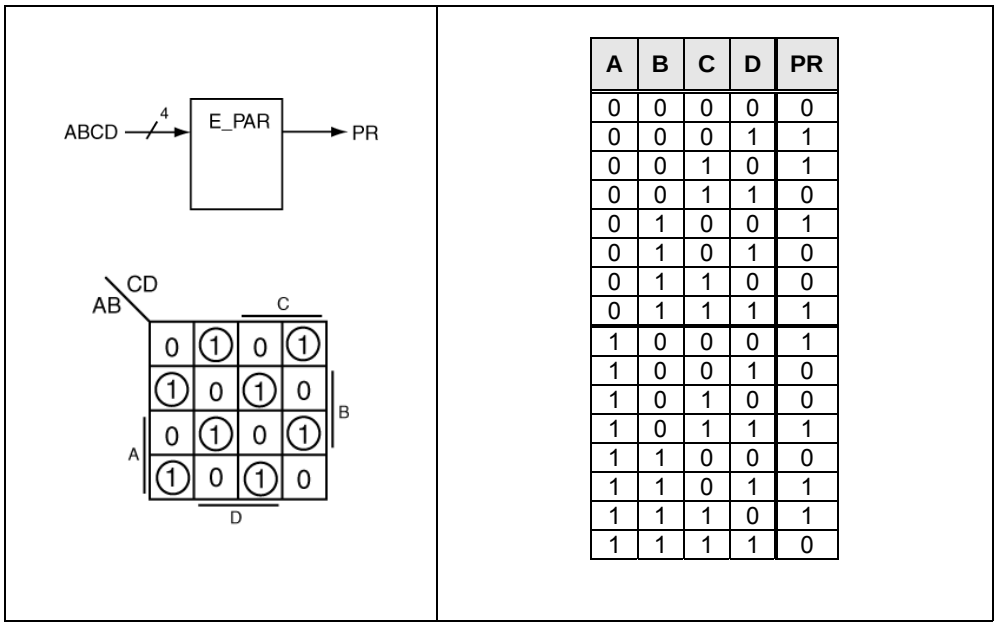
Figure 3-15: A black box model and a VHDL model for a standard 2-bit comparator.

```
entity comp2b is
   port (A,B : in std_logic_vector(7 downto 0);
          EQ : out std_logic);
end comp2b;

architecture process_statement of comp2b is
begin
   c2b: process (A,B)
   begin
      if (A = B) then
         EQ <= '1';
      else
         EQ <= '0';
      end if;
   end;
end process_statement;
```

Figure 3-16: A VHDL model for a simple two-bit comparator.

3.3.9 Parity Generator

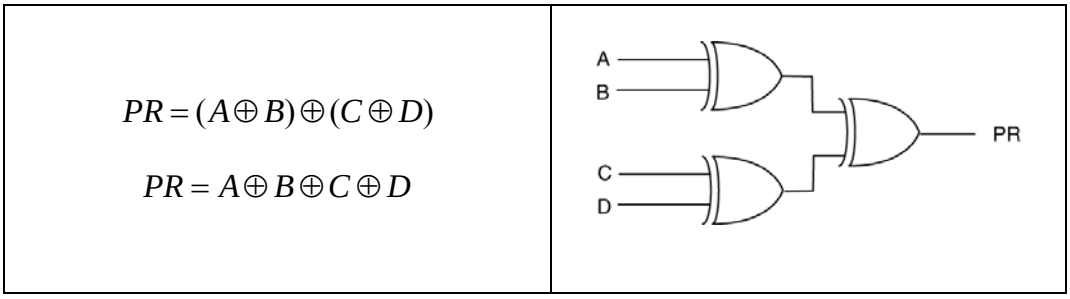
The notion of parity states a characteristic regarding a "set" of bits. If you add up all the bits and there is an odd number of '1' bits, the set of bits has odd parity; otherwise, the bits have even parity. Parity is often in communication protocols to provide 1-bit "error detection". The notion of parity is should bring up the thought of EXOR functions as implied by the K-map in Figure 3-17(a). The truth table in Figure 3-17(b) and the K-map in Figure 3-17(a) describe an even parity generator (look it over and make sure you know why). Figure 3-18 shows the associated Boolean equations and circuitry for Figure 3-17. Note that this circuit is another example of iterative modular design in that we can extend the number of bits in the parity generator by adding more XOR gates.



A	B	C	D	PR
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(a) (b)

Figure 3-17: The truth table and K-map for the 4-bit even parity generator.



(a) (b)

Figure 3-18: The equations and circuit for the 4-bit even parity generator.

3.4 Sequential Circuits

Sequential circuits are circuits that have the ability to "remember" at least one bit. The notion of remembering bits give sequential circuits the notion of having "state". And thus, the notion of finite state machines (FSMs) is born. Keep in mind that the outputs of a sequential circuit depend on the past sequence of inputs to that circuit. The most simple 1-bit storage element in digital design land was the "latch" which was based on cross-coupled NOR and NAND cells. WE consider latches to be "level sensitive" devices.

3.4.1 Flip-Flops

There are three main types of flip-flops in digital design-land: the D, T, and JK flip-flop. Table 3.2 shows the characteristics of these flip-flops and represents all you should know (or be able to figure out real quickly) about these devices. While the D flip-flop is the most popular flip-flop out there, it builds character to know

about all the major types of flip-flops. For your information, a standard Xilinx FPGA generally has about a bajillion D flip-flops included in its architecture. We consider flip-flops to be "edge sensitive" devices.

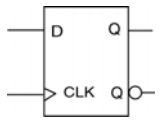
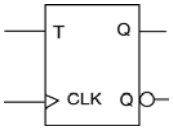
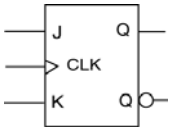
<table>
<tr><th>Type</th><th>Symbol</th><th>Characteristic Table</th><th>Characteristic Equation</th><th>Excitation Table</th></tr>
<tr><td>D</td><td> 
</td><td>
<table>
<tr><th>D</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table></td><td>$Q+ = D$</td><td>
<table>
<tr><th>Q</th><th>Q+</th><th>D</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>T</td><td> 
</td><td>
<table>
<tr><th>T</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table></td><td>$Q+ = T \oplus Q$</td><td>
<table>
<tr><th>Q</th><th>Q+</th><th>T</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table></td></tr>
<tr><td>JK</td><td> 
</td><td>
<table>
<tr><th>J</th><th>K</th><th>Q</th><th>Q+</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
</table></td><td>$Q+ = J\overline{Q} + \overline{K}Q$</td><td>
<table>
<tr><th>Q</th><th>Q+</th><th>J</th><th>K</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>-</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>-</td></tr>
<tr><td>1</td><td>0</td><td>-</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>-</td><td>0</td></tr>
</table></td></tr>
</table>

Table 3.2: The important characteristics of the D, T, and JK flip-flops.

3.4.2 Finite State Machines (FSMs)

FSMs generally serve as controllers, or, circuits that control other circuits. Figure 3-19 and Figure 3-20 shows the two types of FSMs: *Moore* and *Mealy* machines. As you can see from a brief perusing of these figures, these two types of FSMs are more similar than they are different so we'll discuss the similarities first. The terminology used to describe the three basic components of FSM differs widely from source to source but the general function of the three components is necessarily equivalent.

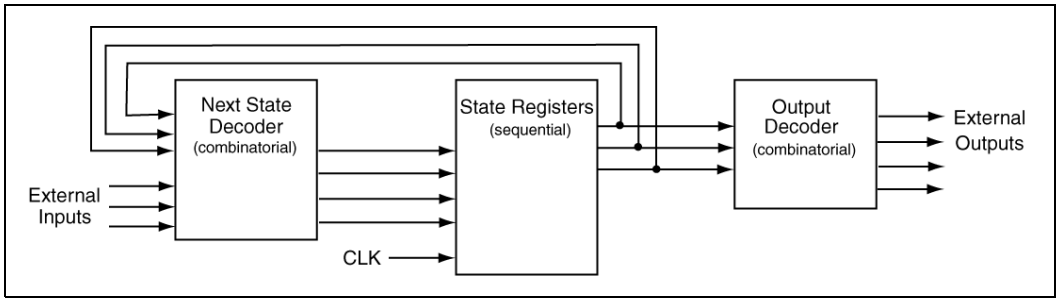


Figure 3-19: Model for a Moore-type FSM.

Although a FSM has lots of internal digital circuitry, we can easily abstract the functionality into three separate blocks: 1) Next State Decoder, 2) the State Registers, and 3) the Output decoder. Table 3.3 provides a detailed description of the individual blocks of Figure 3-19 and Figure 3-20. Understanding the basic

functioning of the blocks is the key to understanding how these blocks interact with each other (it's the system-level thing all over again) and form the FSM.

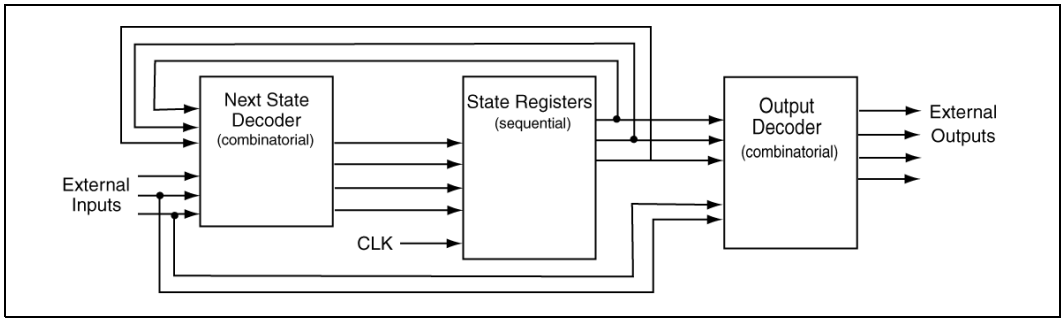


Figure 3-20: Model for a Mealy-type FSM.

Module	Description and Comments
State Registers	The state registers are the memory elements in the FSM. The term *register* in digital land is term that is implies some type of synchronous storage. In the case the flip-flops, each flip-flop can store a single bit of information. The state registers are the only sequential part of the typical FSM; the other two blocks are combinatorial in nature. In other words, the clock signal present in Figure 3-19 only affects the state registers. The state registers store the *state variables* of the FSM. The bits that are being stored in the state register flip-flops determine the state of the FSM. The purpose and function of the state registers is identical for both Mealy and Moore-type FSMs.
Next State Decoder	The next state decoder is a set of combinatorial logic that provides excitation input logic to the flip-flops in the state registers. The next state logic generally has two types of inputs: 1) the current value of the state variables, and, 2) the current value of the inputs from the external world. These two sets of values form what we refer to as *excitation inputs* to the state register flip-flops. Recall that the inputs to the flip-flops are used to determine the flip-flops *next state* (following the next active clock edge). The important thing to notice here is that the next state of the flip-flops, or the next state of the FSM, is a function of both the external inputs and the current present state of the state registers. Once the active clock edge arrives, the flip-flops act on the excitation inputs and the next state becomes the current state. We sometimes refer to the next state decoder as the *next state logic*, or the *next state forming logic*. The purpose and function of the state registers is identical for both Mealy and Moore-type FSMs.
Output Decoder	The output decoder is a set of combinatorial logic that generates the external outputs of the FSM. The only difference between a Mealy and Moore-type FSM is that the Moore machine does not include external inputs to the output decoder; Figure 3-19 and Figure 3-20 show these differences in the inputs to the output decoder blocks. In a Mealy-type FSM, the external outputs are a function of both the state variables and the internal inputs. In a Moore-type FSM, the external outputs are strictly a function of the state variables. Having a fundamental understanding of the differences between a Mealy and Moore-type FSM is integral to understanding and designing FSM-based controllers.

Table 3.3: A detailed description of the three main FSM functional blocks.

The heart of the FSM is the state registers; the heartbeat of the FSM is the clocking signal that controls the state-to-state transitions of the FSM. On each active edge of the clock, the state of the FSM can change. The

excitation inputs to those flip-flops govern the state transitions of each flip-flop in the state registers. The excitation inputs to the flip-flops are the outputs of the next state decoder. The next state decoder's outputs are formed by the logic internal to the next state decoder and are a function of the present state of the FSM and the external inputs. The FSM's external inputs are generally status signals from the outside world. The FSM sends the control signals to the outside world via the output decoder. The external outputs from the FSM are a function of the state variables (Moore-type FSM) or a function of both the state variables and the current external inputs (Mealy-type FSM).

3.4.2.1 The State Diagram

The state diagram is one of many methods used to describe a FSM. The particularly pleasing aspect of the state diagram is that its purpose is to convey meaning and understanding to the human viewer. As you will soon find out, there several approaches available to implement FSMs. The following are the four types of information presented by state diagrams.

1. Various states in the FSM
2. Various state transitions associated with the FSM
3. Input conditions controlling the state-to-state transitions
4. Output values of the outputs associated with the states (and external inputs for Mealy-type FSMs)

This section deals primarily with the transitions associated with a state diagram. Figure 3-21(a) shows a typical (and overly generic) state diagram. The following blurb describes some of the key features of this state diagram.

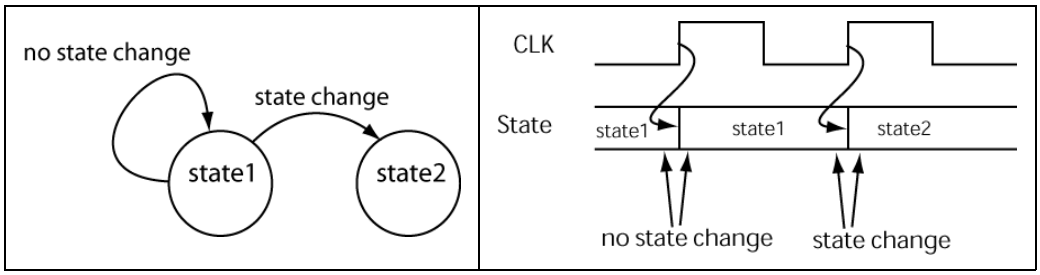
- The terminology we use to describe how a FSM goes from one state to another is a *state transition* or just *transition*. An arrow directed from the source state to the destination state represents the transition between states. Roughly speaking, there are only two possible state transitions in a state diagram from a given state: on the associated clock edge, a transition can occur from 1) one state to another state (indicated by the "state change" label in Figure 3-21 (a)), or, 2) the FSM can remain in the same state (indicated by the "no state change" label in Figure 3-21(a)).

- The state diagram contains no clock signal as you may expect. Even though the system clock is an integral part of a FSM, state diagrams never show a clock signal. The only part of the clock signal we're interested in this the clock edges; the arrows represent what action occurs on the clock edges.

- The two states shown in Figure 3-21 (a) have unique names. In real life, you would want to give these more meaningful names such as something to indicate why the state exists (or what is going on in that state). The state names provided Figure 3-21 (a) give no indication as to how the states are represented when you synthesize the FSM.

- The relation between the timing diagram shown in Figure 3-21 (b) and the state diagram in Figure 3-21(a) is the key to understanding state diagrams in general. When we talk of state, we're talking about all the time in-between the active edges of the clock. In other words, the state bubble essentially represents all the time between any two active edges of the clock. On each clock edge, one of two things will occur; either the FSM transitions to another state or the FSM remains in the same state.

- The concept of Present State (PS) and Next State (NS) is somewhat hard to pin down in a timing diagram such as the one shown in Figure 3-21(b). The problem is that the present state (and hence the next state) is constantly changing as you travel from left to right on the time axis. If you declare one state as the present state, then you can declare the following state as the next state relative to the present state.



(a) (b)

Figure 3-21: The state diagram (a), and the associated timing diagram (b).

3.4.2.2 Conditions Controlling State Transitions

As you would guess from examining the state diagram shown in Figure 3-21 (a), there must be some mechanism that decides on which transition will occur from a given state. Note in Figure 3-21 (a) that **state1** has two arrows leaving the state; so there must be conditions that decide on which transition will actually occur. In the global sense, there are two forms of information that will decide on what transitions the FSM will take from any given state: 1) at least one of the external inputs to the FSM, and, 2) the given state the FSM is currently in (otherwise known as the present state). The second condition is somewhat too general because when we're talking about state transitions, we talk about them from the context of being in one state and transitioning to another state. Each state has its own set of conditions that govern transitions so in terms of state diagrams, we're more concerned on a state-by-state basis as to what external input conditions control the state transitions from a given state. Figure 3-22 shows the way we indicate these conditions. As you can see from Figure 3-22, we list the rules governing transitions by placing the conditions next to the state transition arrows.

1. The conditions associated with the state transition arrows from a given state must be mutually exclusive. This means that there can never be a particular set of input conditions that is associated with two different transitions arrows leaving the same state.

2. The set of conditions associated with a particular state must be complete. If there is a set of conditions from a given state that the state diagram does not cover using the associated state transition arrows, the FSM will once again not know the correct thing to do.

3. If the state lists no conditions with the state transition arrow, this usually means the state transition is unconditional. Once again, it is generally a better ideal to provide a "don't care" of the condition portion of the state diagram.

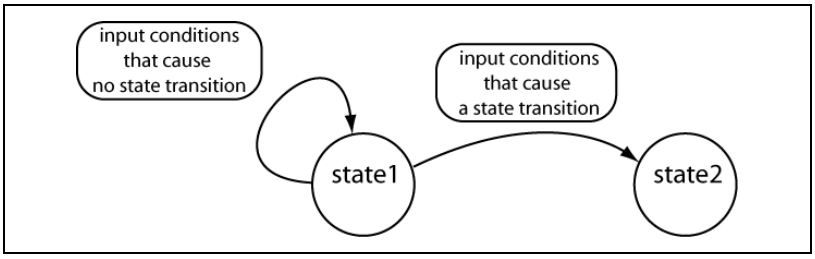


Figure 3-22: How state diagrams indicate the conditions associated with state transitions.

One thing to keep in mind here is that the FSM is a piece of hardware that controls another piece of hardware. The external inputs to the FSM are generally status inputs from the circuit that the FSM is controlling. The

idea here is that, depending on the current status of the hardware that the FSM is controlling, the FSM will transition to one state or another. The thing we haven't mentioned yet is that there are also some external outputs from the FSM, which serve as control inputs to the circuit that the FSM is controlling.

3.4.2.3 External Outputs from the FSM

In that the external input signals serve as status inputs to the FSM, the external output signals directly control some other circuit. Generally speaking, the state diagram will have different states and the control signals output from one state are generally not the same as control signals output from other states.

There are two different types of outputs in a FSM: Mealy-type outputs and Moore-type outputs. Although these two types of outputs are similar in most aspects, particularly in their controlling functions, they have one major difference. The outputs Moore-type outputs are a function of the state variables only while the Mealy-type outputs are a function of both the state variables and the current external inputs.

What we're concerned about in this set of section is how we're going to represent the Mealy and Moore-type outputs on the state diagram. The key to any state diagram is the legend that tells the viewer how to interpret what they're looking at. So if you deviate from the approach for representing state diagrams provided here, be sure to provide a detailed legend in order to appease the FSM Gods.

Since Moore-type outputs are a function of the state variables only, we represent them by placing their values inside of the state bubble. Figure 3-23 shows a state diagram that uses this approach. There can be any number of outputs represented inside the bubble. A comma usually delineates different outputs but you can use whatever method you choose so long as it is clear. Don't be afraid to increase the size of your bubble on your state diagram in order to include all the outputs. Clarity and readability takes home the prize when drawing state diagrams.

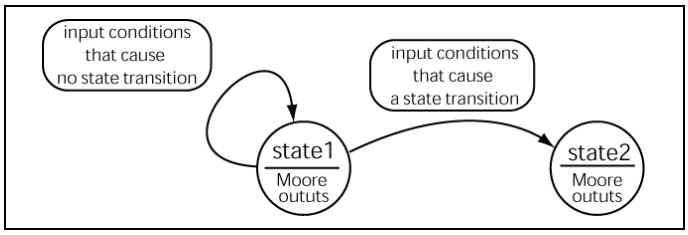


Figure 3-23: The State Bubble with associated Moore outputs.

We do not represent Mealy-type outputs inside of the state bubble because they are a function of some combination of the external inputs as well as the state variables. To account for these characteristics in a state diagram, we list the outputs next to the external inputs associated with the individual state transition arrows and differentiated by the addition of the forward slash. Figure 3-24 shows an example of this approach. Once again, if a particular FSM has multiple Mealy-type outputs, these should be represented with either a comma separated list or something equally as readable.

There is a massively important feature shown in Figure 3-24 that can sometimes go without notice. Note that there are two sets of Mealy outputs shown in Figure 3-24 because there are two transitions from **state1**. The arrows are associated with the state transitions, which are an exclusive function of the current external inputs. Since the Mealy-type outputs are based a function of the external inputs, they are represented by placing them next to the particular external inputs and associated with a given state transition arrow. But… ***the listed condition of the Mealy-type outputs is always associated with the state the arrow is leaving*** (and not the state the arrow is entering). In addition, it should come as no surprise that you can represent both Mealy and Moore-type outputs in the same state diagram as is shown in Figure 3-25.

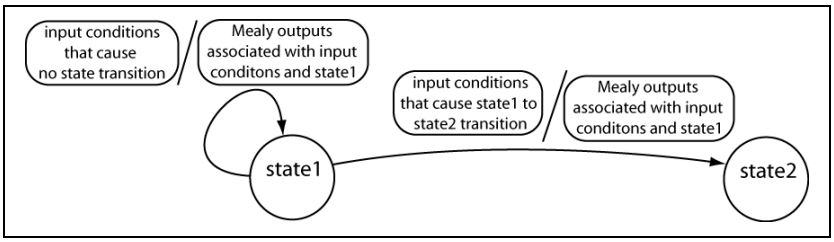


Figure 3-24: Representing Mealy-type outputs in a state diagram.

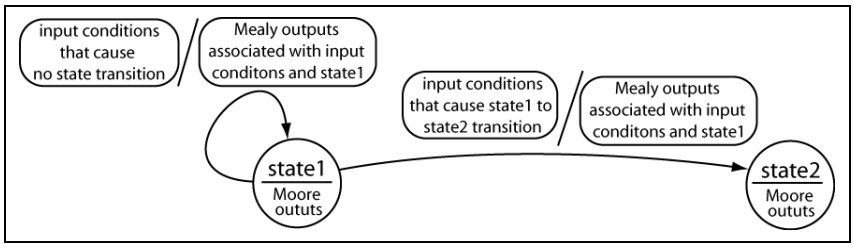


Figure 3-25: A state diagram that has both Mealy and Moore-type outputs.

Figure 3-26 provides a quick overview of the relation between the FSM black box and the example state diagrams we've been working with in this section. What you should be gathering from this diagram is that properly designed state diagrams have a particular structure that is supported by a particular symbology.

- Arrows represent state transitions.
- The FSM has external inputs that govern the state transitions.
- The external inputs are listed with each transition arrow
- Moore-type outputs are listed inside the state bubbles since they are only a function of state
- Mealy-type outputs are listed with the internal inputs (and hence the state transitions) since they are a function of both the present state and the external inputs.

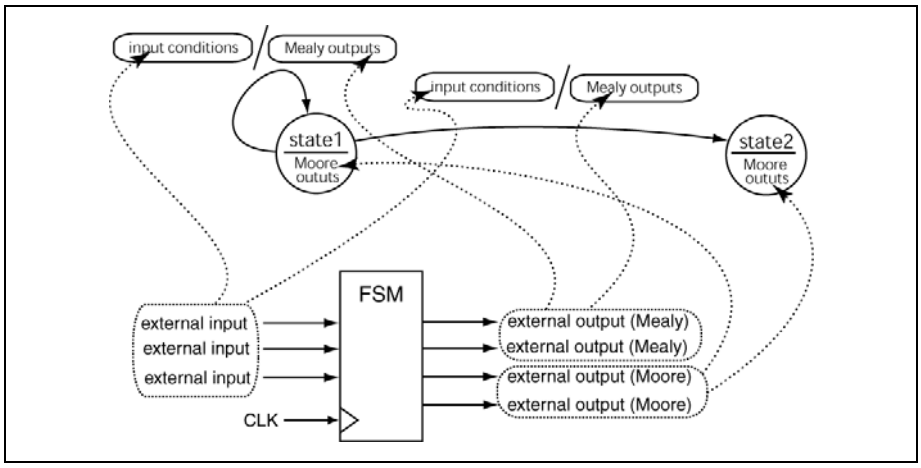


Figure 3-26: The relation between the state diagram and the high-level FSM.

3.4.2.4 Finite State Machines Modeling in VHDL

There are generally many ways to model a FSM using VHDL. Generally speaking, modeling FSMs with VHDL is primarily grunt work; the real engineering is in generating the state diagram.

```
entity fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                Z1,Z2 : out std_logic);
end fsm;

architecture my_fsm of fsm is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin

   sync_proc: process(CLK,NS,RESET)
      begin
        if (RESET = '1') then PS <= ST0;
        elsif (rising_edge(CLK)) then PS <= NS;
        end if;
      end process sync_proc;


   comb_proc: process(PS,X)
      begin
         Z1 <= '0'; Z2 <= '0';
         case PS is
            when ST0 =>    -- items regarding state ST0
               Z1 <= '1';  -- Moore output
               if (X = '0') then NS <= ST1; Z2 <= '0';
               else  NS <= ST0; Z2 <= '1';
               end if;
            when ST1 =>    -- items regarding state ST1
               Z1 <= '1';  -- Moore output
               if (X = '0') then NS <= ST2; Z2 <= '0';
               else  NS <= ST1; Z2 <= '1';
               end if;
            when ST2 =>    -- items regarding state ST2
               Z1 <= '0';  -- Moore output
               if (X = '0') then NS <= ST3; Z2 <= '0';
               else  NS <= ST2; Z2 <= '1';
               end if;
            when ST3 =>    -- items regarding state ST3
               Z1 <= '1';  -- Moore output
               if (X = '0') then NS <= ST0; Z2 <= '0';
               else  NS <= ST3;  Z2 <= '1';
               end if;
            when others => -- the catch all default case
               NS <= ST0; Z1 <= '0'; Z2 <= '0';
         end case;
      end process comb_proc;

   with PS select
      Y <= "00" when ST0,
           "01" when ST1,
           "10" when ST2,
           "11" when ST3,
           "00" when others;
end my_fsm;
```

Description of FSM's interface (I/O)

Declaration of VHDL type used to represent the states of the FSM.

The "synchronous" process to control FSM parameters associated with the state variable representation (storage elements).

The combinatorial process to handles output decoding and next-state assignments.

All outputs are assigned initial values at start of process.

The "cases": one case for each FSM state (and a default case too).

Moore output assigments are outside conditional statement

Mealy output assignments and next state assigments are inside the conditional statement.

Concurrent statement to provide desired output assignments based on FSM states; required if the state variables are used as FSM outputs.

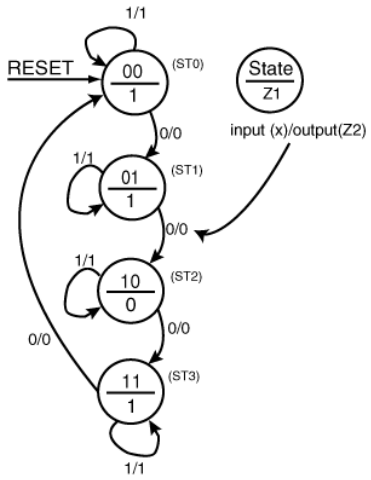


Figure 3-27: The VHDL FSM model cheat sheet with explanations.

3.5 Digital Design Approaches

There are three approaches to performing digital design; any possible digital design you do necessarily fits into one of these approaches. The following outline and Table 3.4 describes these approaches.

1) Brute Force Design (BFD): Also known as iterative design. This was the first design approach we worked with and is based on assigning outputs to every possible input combination via a truth table.

2) Iterative Modular Design (IMD): This was the second approach to design we worked with. Most appropriately, IMD would be included as a subset of modular design, but we're opting to call it a design approach all its own. This design approach allowed us to bypass the truth table approach of BFD and enabled us to create mildly complex circuits such as the ripple carry adder.

3) Modular Design (MD): In this approach, we were most interested in drawing bunches of dark boxes to model our designs. We also drew boxes within boxes within boxes, which we labeled as hierarchical design[1]. Recall that we've been claiming all along that hierarchical design is massively powerful; now we'll state that modular design of any type is massively powerful.

Design Approach	Pros	Cons
Brute Force Design	Really straight forward	Limited by truth table size
Iterative Modular Design	Straight forward	Not applicable to all designs
Modular Design	Massively powerful	Requires a working brain

Table 3.4: Matrix explaining why Modular Design can save the world.

Modern digital design consists primarily of Modular Design. You do modular design by plopping down dark boxes and connecting them up in intelligent ways that solve your given problem. The dark box diagrams are of course a form of modeling. The black box models convey various levels of information regarding the digital circuit, but most importantly, someone can take you model and actually convert it to a working circuit.

There are only a few rules you need to follow when doing modular design.

- **Be clear and concise:** A messy dark box model or circuit diagram is a tragedy that hinders the transfer of information. Strongly consider using a ruler if you're modeling by hand.

- **Label everything:** Make sure the reader of your model does not need to make any assumptions about anything.

- **Provide a definition for all dark boxes:** Black box modeling facilitates the notion of modern digital design. Every box you use in your model should either be clearly defined somewhere (such as at another level) or be a standard digital "box". There are many standard digital "boxes" out there. If you call out one of these boxes in your black box models, everyone will know what you're talking about and there is no need to define it at a lower level. The catch here is that you must use these boxes in the exact way there were defined; if you don't, people will not know what you're trying to model. Table 3.5 shows a few examples of proper black box usage.

[1] Thus, hierarchical design can be considered a form of modular design.

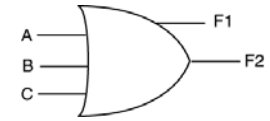
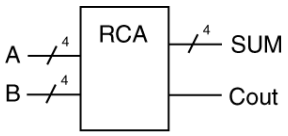
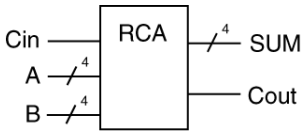
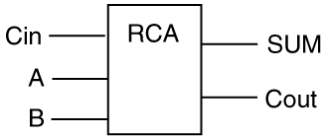
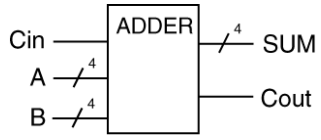
Model	Comment
	This sort of looks like a 3-input OR gate, but having two outputs makes it non-standard. Being non-standard, it's a mystery how the circuit assigns the outputs. This is a bad model. To make it valid would require that it be defined somewhere so we all know what it is.
	This is a true digital box. Since we know what an RCA is, and the inputs and outputs of the box labeled RCA match what we know about RCAs, we know exactly how it works. This is a valid model and there is no need to define it further.
	This is also a true digital box. If you replace the HA in a RCA with a FA, you'll have the extra carry-in input as is listed in this model. Having this input is handy and often useful. This is a valid model.
	This circuit has the RCA label, but since we know RCAs to have multiple inputs (bundles) for the addend and augend, we're left scratching our heads. You could assume it's a RCA but you could be wrong. This is an invalid model.
	This has all the correct inputs for an RCA, but since it has the ADDER label, we can't assume we know exactly what this box is doing. This is an invalid model. You could make this model valid by providing a definition for the ADDER somewhere in your design.

Table 3.5: Some good and bad example of standard digital dark boxes.

One of the underlying themes in digital design is the use of modularity. To put this statement in other terms, you can subdivide even the most complex digital circuit into a set of the relatively few standard digital circuits. However, if you take that previous statement and look at it in the opposite way, you can create complex digital circuit designs by connecting a set of standard digital circuits in an intelligent way.

The general approach to becoming an efficient digital designer is to design on as high of level as possible. In terms of the three design techniques, we've discussed thus far, that means you should always aim for the modular design approach, which necessarily incorporates all of your previously designed digital modules. If you find yourself reinventing the wheel in each of your digital designs, be aware that there are better approaches. If you need to use one of those standard digital circuits in your particular design, go back and reuse as much of your old design as possible. One of the many great attributes about VHDL is its support of modular design and module reuse.

3.6 VHDL Introduction: Modeling Digital Systems

VHDL is an acronym that for – **V** (Very High-Speed Integrated Circuit) **H**ardware **D**escription **L**anguage. In other words, VHDL is a *hardware description language.* We use VHDL to model digital circuits and systems. The definitions of models and systems change depending upon the context in which they are used. In my

experience, many people use these terms loosely and look down upon you if your definition is different from their definition. Here are the definitions for these two terms, as we will use them for CPE 233.

> ***Digital System*:** any circuit that processes or stores information. (As you can see, this is a loose definition, so loose that it includes a digital circuit such as a NAND gate, and also a complex microprocessor).
>
> ***Model*:** A model is a definition or description of something that presents a particular level of detail. (Once again, this definition is really loose. In actuality, there is no absolute model of something; the correct model is dependent upon the context in which it is used.)

Modeling circuits using VHDL has many advantages. At first, it seems like a pain-in-the-arse to learn the syntax for yet another computer language. The natural tendency is to stay towards using a schematic capture program to model circuits. This has severe limitations and is rarely done anymore except by people too close to retirement to learn a better way to do it. In actuality, modeling in general, and modeling digital circuits using VHDL has the following advantages.

- The different level of models supports the understanding of complex digital systems. By abstracting circuits to higher levels (such as modeling them as black boxes) allows you to ignore some of the less relevant details at the lower levels.
- VHDL presents a formal and unambiguous way to model circuits. This allows everyone to "speak the same language" when dealing with digital circuits.
- VHDL models allow for the verification of circuits using simulation. VHDL models can be input to various simulators to provide output to various input stimulus.
- Most importantly, software can use VHDL models to automatically generate, or synthesize, digital circuits. The synthesize circuit can then be downloaded onto various types of programmable logic devices (PLDs) or be used to generate the masks required to create VLSI circuits.

3.6.1 VHDL Overview

The best place to start when looking at VHDL are the "easy" things. If you commit the following things to memory, you'll save yourself a lot of time down when you write your own VHDL code.

VHDL Invariants: These are ideas in VHDL that never change:

- Case Sensitivity: not case sensitive (identifiers between quotation marks usually are).
- White space: white space is ignored
- Comments: anything after a "- - " (two consecutive dashes) is ignored
- Parenthesis: few requirements; use for code clarity
- Statement Termination: semicolon
- IF, CASE, and LOOP statements:
  - Every IF statement has a "then" associated with it
  - Every IF statement has an "end if" statement associated with it
  - Other branches of the IF statement are "elsif" (not else if)
  - Every CASE statement has and "end case" statement associated with it
  - Every LOOP statement has an "end loop" statement associated with it
- Coding Style: freedom with case sensitivity, white space, and parenthesis require that the programmer put effort into making the code readable.

The verbage that follows represents an overview of VHDL in order to remind you what you should already know. The approach taken in this introduction is to exploit the similarities between the concepts and ideas in

both VHDL and a higher-level language such as C or Java. Although there are many similarities, always keep in mind that VHDL is used to model hardware and is thus much different from programming a higher-level language.

A good programming practice is to structure programs in a modular fashion. Most higher-level languages support the notion of modularity with constructs such as function, subroutines, and procedures. A program written in a modular fashion is easier to read, understand, modify, maintain and the modules support easy reuse. A step beyond this level includes the creation of libraries and such. The same ideas apply to VHDL but the lingo is a bit different. Modularity in VHDL allows the modeling of circuits at different levels. Table 3.6 shows some of the analogous concepts and terms for VHDL and higher-level languages.

Higher Level Languages		VHDL
functions, subroutines, procedures	⇔	**entity**
function prototype- function declaration	⇔	**entity** declaration
formal parameter, arguments	⇔	input signals
return values	⇔	output signals
function definition	⇔	**architecture**

Table 3.6: A comparison of Higher-level language and VHDL vernacular.

What drives the general paradigm behind VHDL is the fact that someone needs to design and/or simulate hardware. This is paradigm is different from that of higher-level languages. In higher-level languages, the statements that appear in a program execute sequentially. Generally speaking, the statements are to be executed by the processor and the processor can only do one "thing" a time (namely, execute statements). Table 3.7 shows two sets of statements that look similar. The big difference is that the higher-level language statements execute sequentially while the VHDL statements execute *concurrently*. In other words, it's easy to imagine the listed VHDL statements execute concurrently once you visualize the circuit associated with the VHDL statements listed in Table 3.7. Figure 3-28 shows an equivalent circuit for the VHDL statements of Table 3.7.

higher level language statements	VHDL statement
`A = B + C;`	`A <= B AND C;`
`D = E + F;`	`D <= E AND F;`
`G = H + I;`	`G <= H AND I;`

Table 3.7: Some similar statements in C and VHDL.

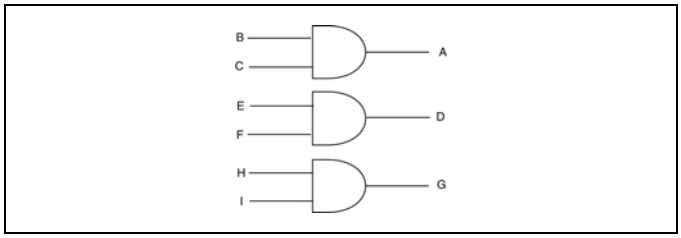


Figure 3-28: An equivalent circuit for the VHDL statements shown in Table 3.7.

Remembering that the focus of VHDL is to design hardware, it should not be surprising that notion of "concurrency" is important. That being the case, there are four major types of concurrent statements in VHDL:

1) Concurrent signal assignment
2) Conditional signal assignment
3) Selective signal assignment
4) Process statements

The main assignment operator in VHDL is the *signal assignment operator*: "**<=".** Figure 3-29 provides a quick example of circuit implemented in using three different types of VHDL statements.

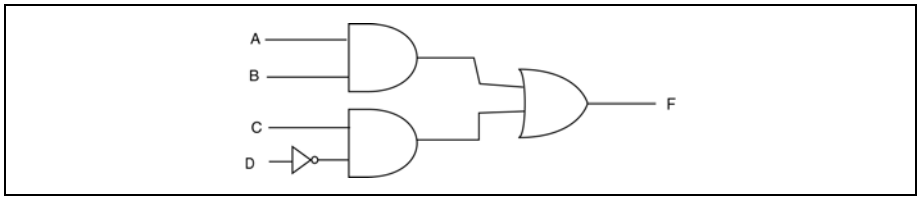


Figure 3-29: A quick and meaningless example.

Solution: Table 3.8 shows three functionally equivalent VHDL models for the circuit example of Figure 3-29. Note that the three architectures use the same entity.

entity declaration	`entity my_ckt is`
	`  Port ( A,B,C,D : in std_logic;`
	`  F : out std_logic);`
	`end my_ckt;`

concurrent signal assignment	`architecture ckt1 of my_ckt is`
	`begin`
	`  F <= (A AND B) OR (C AND (NOT D));`
	`end ckt1;`

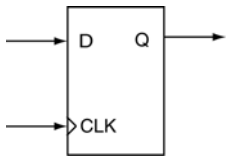
conditional signal assignment	`architecture ckt2 of my_ckt is`
	`begin`
	`  F <= '1' when (A = '1' AND B = '1') else`
	`  '1' when (C = '1' AND D = '0') else`
	`  '0';`
	`end ckt2;`

selective signal assignment	`architecture ckt3 of my_ckt is`
	`  signal ABCD : std_logic_vector(3 downto 0);`
	`begin`
	`  ABCD <= (A & B & C & D);`
	`  with (ABCD) select`
	`  F3 <= '1' when "1000" | "1001" | "1100" | "1101" | "1110" | "1111",`
	`  '0' when others;`
	`end ckt3;`

Table 3.8: Example VHDL models of the meaningless example.

3.6.2 Basic Memory in VHDL

The general approach to learning about how VHDL models simple storage elements is to examine a simple model of a D flip-flop. Figure 3-30 shows the preferred VHDL model of a D flip-flop.



(a)

```
-------------------------------------------------
-- Model of a simple D Flip-Flop
-------------------------------------------------
entity d_ff is
    port ( D, CLK : in std_logic;
                 Q : out std_logic);
end d_ff;

architecture my_d_ff of d_ff is
begin
   dff:  process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process dff;

end my_d_ff
```

(b)

Figure 3-30: The black box diagram and VHDL model for a D flip-flop.

The D flip-flop is best known and loved for its ability to store (save, remember) a single bit. The way that the VHDL code listed in Figure 3-30 is able to store a bit is not obvious, however. The bit-storage capability in the VHDL is *implied* by the both the VHDL code and the way the VHDL code is interpreted. The implied storage comes about as a result of not providing a condition that indicates what should happen if the **`if`** condition is not met, otherwise known as a "catch-all" condition. In other words, if the **`if`** condition is not met, the device does not change the current value of **Q** and therefore must "remember" that current value.

The remembering of the current value, or state, constitutes the famous bit storage quality of a flip-flop. If you have not specified what the outputs should be for every possible set of input conditions, there will be conditions where the changes in the output are not defined. In these cases, the option taken by VHDL is not to change the current output. By definition, if the inputs change to an unspecified state, the outputs must remain unchanged because it is not defined how they should change. This is the mechanism, as strange and interesting as it sounds, is how VHDL *induces* memory. This is also why we are careful to always provide a "catch-all" condition for our previous VHDL models. It was the inclusion of the catch-all condition in our models that assured us that we were not inducing memory elements.

The typical method used to provide a catch-all condition in case the **`if`** condition is not met is with an **`else`** clause. Generally speaking, a quick way to tell if you've induced a memory element is to look for the presence of an **`else`** clause associated with the **`if`** statement. Once again, it is the else statement that provides the "catch-all" characteristic of the model.

3.7 A Detailed Computer-ish Design Example

The overall CPE 233 course objective is the understanding of basic computers and their basic low-level programming. More specifically, we'll be study the basic computer on a block level in an effort to understand its basic functioning. We'll use this understanding to extend the basic computer design and to obtain a good grasp of the hardware limitations of computers. A firm understanding of these areas will necessarily make us better assembly language programmers. If we are good assembly language programmers, then we'll necessarily understand the underlying architectural elements of a computer.

Once again, CPE 233 comprises of two main parts: computer architecture and assembly language programming. It's still a mystery regarding the best approach to working with these topics in a 10-week course, but with every course offering, the approach is sure to become better. The main issue is that master of

assembly language programming requires a complete understanding of underlying computer architecture. But, the main goal of designing a computer is to provide a set of assembly language instructions that are useful to programmers (or compiler writers) while being relatively efficient in the process. It's sort of a chicken and egg problem. From my experience, working with one aspect then the other is not optimal. The best approach seems to be to simultaneously introduce both computer architecture and assembly language programming concepts.

A computer is a digital system, which implies that it is comprised of a bunch of gates and flip-flops that someone connected in some intelligent manners. From a higher level, a computer is nothing more than a special connection of all the standard digital circuits you've learned about up until now. A computer is basically no different than any of the other digital systems you've worked with except that it can become more complex. But then again, the complexity comes from the sheer amount of simple elements in the circuit and not the elements themselves[2].

Figure 3-31 shows the basic model of a computer that we'll be working with in this course. As you can see from Figure 3-31, a computer is comprised of three main components: the central processing unit (CPU), memory, and input/output (I/O). One thing to keep in mind is that many devices in our modern world fit nicely into the model of Figure 3-31. A wristwatch is as much of a computer as is the computer on your desk (as is the microwave oven in your kitchen). We can characterize each of these devices by a CPU constantly executing programs stored in memory while interfacing with the outside world via the I/O.

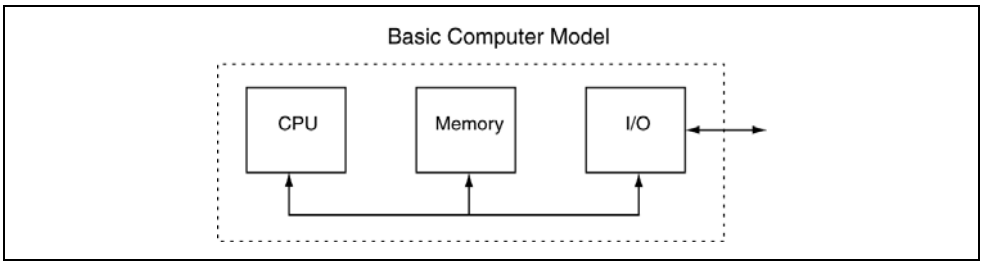


Figure 3-31: A basic model for a computer system.

Figure 3-32 shows the diagram we're more interested in for the initial part of CPE 233. This model shows the two basic components of the CPU (central processing unit): the *control unit* and the *datapath*. Although the memory and I/O blocks of Figure 3-31 are both important and highly interesting, time constraints limit our approach to only a general overview of the topics associated with these blocks and we'll deal with those topics later. We briefly describe these components below. But before we look at this, let's back up a step.

The name "CPU" has actually says more than you would first imagine. As you would guess by know, the only thing a computer really does is tweak data, or more appropriately put, *process* data. There are many ways to do "data processing" in a computer. The most general approach is to perform all of the processing in one location, or more appropriately stated, using one piece of hardware. Although it would be possible to perform processing at any location in hardware (or multiple locations), generally it is done in one location. This one location in the actual hardware is where the term "central" comes from in the term "CPU".

The notion of the "central" processing unit came from days where hardware was massively expensive, both on the discrete level and on the silicon level. Things are slightly different these days though. As the underlying IC fabrication techniques improved and allowed for smaller digital circuitry, the required processing in computers is harder to describe as "central". Having more processing units increases the overall throughput of the computer, which allows more advanced features on the devices in which these ICs appear. Where would the world be now without a device that allows you to call your buddy and watch a stream of video data all on the same device and at the same time? Now that's progress!

[2] Once again, the key to understanding complex issues such as computers is to divide the associated digital circuitry into more manageable blocks. This form of abstraction is absolutely required because even the simplest computer is arguably complex. Lucky for us that VHDL structural modeling fully support this flavor of abstraction.

The model for performing processing shown in Figure 3-32 is one option of many and we generally use it to describe the basic operation of computers. In general, you could increase the processing power of a computer by including multiple CPUs but this would generally increase the complexity and raise the cost of a computer. The processing power vs. cost is one of the classic tradeoffs in computer architecture.

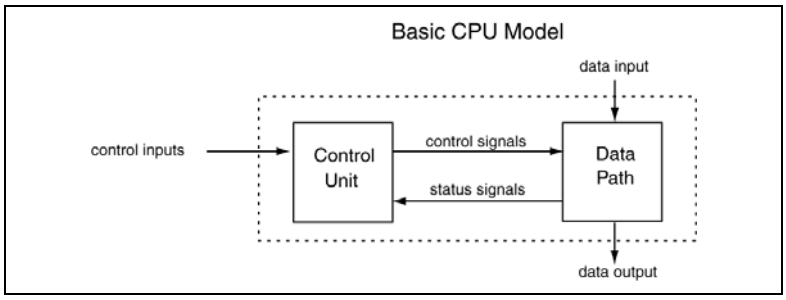


Figure 3-32: A basic model for a CPU.

Figure 3-32 shows that the two main components in the CPU are the control unit and the datapath; here is a brief introductory description of these functional blocks.

The Control Unit: This unit makes certain data processing operations occur in the datapath block. In other words, the control signals from the control unit inevitably activate various operations within the datapath. The control unit generally acts upon the state of both control inputs from outside the CPU and the status signals from the datapath. The control unit is essentially some type of finite state machine (FSM) which is why the first third of this course deals with FSM design. The control unit output signals are generally some combination of Mealy and Moore outputs.

The Datapath: This unit contains a mix of both sequential and combinatorial logic. We define datapaths by their internal registers and the operations we can performed on the data stored in those registers. We arrange the internal circuitry in such a way as to do something useful under the control of the control unit. These "useful" operations occur on the input data, which includes data from memory or the outside works. The results of these operations return to memory or head back to the outside world (I/O). Datapaths generally contain registers to temporarily store data. The internal workings of the datapath can be somewhat complicated but we'll be developing ways to both understand and control this complexity.

The final word here is that we have ten weeks to introduce and discuss the basic building blocks of a computer and learn to program that computer using assembly language. For some people, CPE 233 represents the only exposure to these topics that will be formally provided in their entire academic experience (namely, the EE's taking this course). But the good news is that even though this exposure to these concepts is brief, it's adequate to understand the basic operation of computers as well as basic programming concepts. This is because, generally speaking, most computers operate using the same basic principles. Once you understand these principles in the context of one particular computer design, it's relatively easy to transfer this knowledge to help you understand any computer that you may face in your computer career.

3.7.1 Moving Down the Datapath

The datapath is the bit-crunching heart of the central processing unit (CPU). As was mentioned earlier, the datapath is a giant circuit that is filled with such a great number of simple devices that it becomes somewhat complex to study if your examination is at too low of a level. These simple devices include both sequential and combinatorial circuitry, which interconnects in some intelligent and organized fashion that produces the desired result.

One term that we commonly use to describe many aspects of computers, we also use to describe the basic organization features of the datapath (or any computer circuitry for that matter). The term *architecture* appears often in computerland to describe *the individual modules of a circuit and the connection between the modules*. For example, we could describe the diagram in Figure 3-32 as showing the *architecture* of the CPU. We'll not only be studying the basic architecture of CPUs, we'll also be going a level down in the hierarchy and studying the features that allow the architecture to generate useful results.

Keep in mind that the word architecture is general enough to apply to any block diagram representation of a circuit. The word architecture is not specific to any one level in the underlying hierarchy. In other words, if someone were to ask you to describe the architecture of something, you better first arrive at an agreement to the level of detail that is expected of your description. The idea of general terms such as "architectures" should be no big deal to you: you've experienced all of these concepts before with the term "model". These terms are synonymous.

The most common feature of the lower-level datapath circuitry is nothing more than a simple *register*. You currently have experience working with registers in your work with counters and shift registers. What you need to know for now is that a register is a means to store an arbitrary amount of data; this data is generally stored in a parallel manner (which means data is stored in all the individual devices on a common clock edge). The general approach that we've always taken in the digital design world is to abstract to a higher level to make things more understandable.

Datapaths are best described by the registers they contain and the operations that can be performed on the data in those registers. A register has the ability to perform various operations on the data they store. We refer to these operations to as *elementary operations* and include exciting things such as shifts, loads, etc. When such an elementary operation is performed on data stored in a register, it is referred to as a *microoperation*. Examples of microoperations include the transfer of data between registers, incrementing or decrementing the value stored in a register, or shifting the contents of register etc. Since we necessarily perform microoperations on registers, they we also perform these operations in parallel to the given set of bits in the register. Once again, an elementary operation is a function an individual register can perform. Moving up a level, a microoperation provides a formal name to the process of moving registered data around. In the current context, microoperations officially transfer data through and around the datapath.

3.7.2 Register Transfer Language

As you can imagine, the design and study of computers can be become quite complicated. In an effort to reduce some of this complication, we continue up the digital hierarchy by developing a special notation that describes the operations that take place in a given datapath. The data processing operations that occur inside the datapath are, generally speaking, operations on registers. In other words, we can describe these operations as the movement of data from one resister to another register. We aptly refer to these operations to as *register transfer operations*.

A register has the capability of performing one or more *elementary operations* on data. Such operations include loading, adding, shifting, etc. We refer to these elementary operations performed on the data present in the registers as *microoperations*. An example of microoperations includes transferring data from one register to another, or adding the contents of two registers and storing the results in some other register. We refer to the shorthand notation we use to describe microoperations to as *register transfer notation* and falls into the broad category of a *register transfer language*. As you'll eventually find out, there are many different syntaxes used to describe register transfer operations. Although these syntaxes look different, they all represent a similar set of operations. The important thing is that a single assembly language instruction does nothing more than perform a pre-determined set of microoperations.

In general, we use RTL to describe the architecture and bit-crunching capabilities of the datapath. RTL provides a method to describe the bit-crunching functions of the datapath using a shorthand notation. Up until now, the only method we've had to describe a non-FSM circuit was with relatively low-level VHDL models. While this is an excellent choice, it is not as expressive as RTL can be if you properly apply the RTL. The

control unit provides signal that output a given sequence of microoperations, which subsequently generate a useful and meaningful transit of data through the datapath circuitry.

3.7.3 Control Unit/Datapath Example using a Universal Shift Register

Digital design primarily uses FSMs to control other circuits. It is possible to make a FSM that will output a desired counting sequence (namely counters), but that is more an academic exercise rather than a real-world application. The end product in CPE 233 is to be able to implement real-world applications; specifically a controller for the datapath circuitry of a computer. The problem below attempts to present a real design problem in the context of the control unit/datapath model. The application is somewhat corny, but the underlying design and the steps required to implement the design are nothing short of enlightening.

Example 3.1

Design a circuit that drives an 8-LED display (similar to the ones in on the development boards in the laboratory) in the following manner. The display should have only one LED at a time lit. The lit LED should shift towards the left with on the system clock edge until it arrives as the left-most LED. At this point, the lit LED should then start shifting towards the right until it hits the right-most LED and starts shifting left again. The circuit is controlled by a shift enable input DE. When this input is asserted, the circuit resets itself (the right-most LED lit) and initiates the shifting sequence. When the DE input is not asserted, the LED output stops shifting and remains in the final state it was in before the DE input was unasserted. Implement the design using less than four flip-flops.

The Long-Winded Solution: The obvious solution to this problem is to use a one-hot encoded FSM and use the state variables to drive the output displays. However, such a solution that would be too simple, which is why we place a limit on the number of flip-flops in this design. The better solution is to continue with the theme of CPE 233 and fit the design into the model in Figure 3-33 (although this time, we're not designing a CPU, the model is still valid). The approach we'll take for solving this problem is to first design the circuit for the datapath and then design a circuit that will control it.

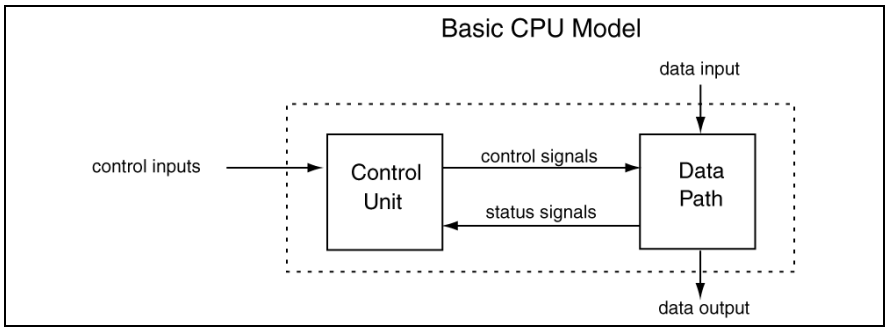


Figure 3-33: The basic model used much too often in CPE 233.

The first step is to pull from your vast digital knowledge and come up with some circuitry for the datapath that will be able to implement the given problem. If you think about it, we can characterize the moving back and forth of the lit LED as a shifting action. Since this shifting action is synchronous with the clock, the design cries out heavily for using a shift register. But didn't shift registers only shift in one direction? A universal shift register that can handle most of the listed functions required by this problem: it shifts left or right, it has a parallel data loading inputs, or it holds the current value. Let's work with this device first.

We base the universal shift register on a simple concept. In other words, we can decompose a shift register down to its most basic component, which we refer to as a shift register cell. As you would guess, this cell is nothing more than a storage element that we usually model as a D flip-flop. Let's first look at a simple shift register (SR) element (a single bit) and expand this outwards to create a universal shift register (USR).

Consider a SR element that is somewhere in the middle of a set of SR elements. For reference, we'll refer to this as the m^{th} SR element in a chain of n elements. Figure 3-34 shows a diagram of such a SR. As you can see the heart of the shift register is the D flip-flop. The circuit in Figure 3-34, although highly interesting, is not overly functional: we'll need to modify it in order to make our datapath do the right thing. The problem with this circuit is that is only shifts data from left to right, which is short of the requirements for this problem. Our approach to this problem will be to modify a single SR element and form a more functional shift register from a string of modified elements.

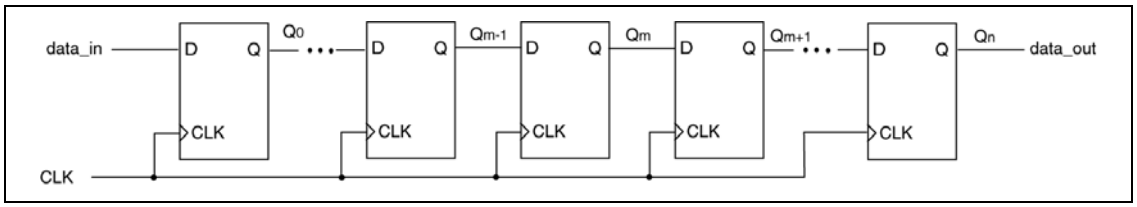


Figure 3-34: A typical n element shift register.

The solution required for this problem is to shift both left and right, do a parallel load, and do nothing (hold the current state of the SR element. To put the requirements in other words, we need to be able to *select* the data that will be loaded into each SR element on each rising clock edge. The key to this circuit lies in the word "select". Anytime you hear this word in digital design land, you should think of "data selector" or more appropriately put as MUX. What we want is to be able to select the data that will be loaded into each SR element and a MUX will do the trick for us. Figure 3-35 shows the final circuit.

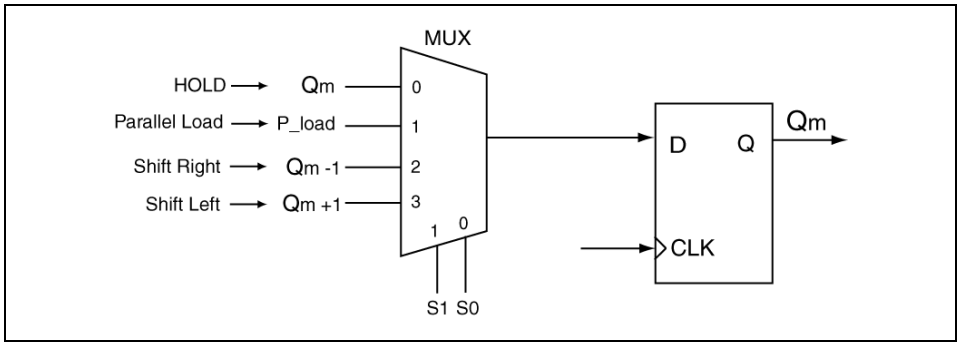


Figure 3-35: A SR element with an attached MUX for data selection.

Figure 3-35 shows that the all the functionality we need to implement the datapath circuit can be generated by applying the correct signals to the MUX. All we need to do from here is to put a bunch of these in a row. For the circuit shown in Figure 3-35, the Q_{m+1} SR element is the D flip-flop to the right of the Q_m element while the Q_{m-1} SR element is the D flip-flop to the left of the Q_m element. By selecting the input that will be loaded to the Q_m SR element, you can effectively implement a shift-left or shift-right functionality. The P_load input provides a means to input a value from the outside world that is not currently in any of the SR registers. Table 3.9 summarizes the functionality of the SR element.

S1	S0	D	Comment
0	0	Q_m	hold
0	1	P_load	parallel load
1	0	Q_{m-1}	shift right
1	1	Q_{m+1}	shift left

Table 3.9: Summary of the SR element functionality.

OK, now that we have our SR element designed, how are we going to implement it? It looks like a lot of circuitry! But alas, VHDL will save our butts once again on this one. The design of a universal shift register using VHDL is straightforward and instructive. It's not too much more complicated than just a simple D flip-flop; or to put it another way, if you understand how a D flip-flop is generated using VHDL, you'll easily understand the VHDL implementation of a universal shift register. Figure 3-36 shows the VHDL model for the universal shift register.

```
-----------------------------------------------------------------
-- datapath for LED bounce display project
-----------------------------------------------------------------
entity univ_sr is
    port (    SEL : in std_logic_vector(1 downto 0);
           P_LOAD : in std_logic_vector(7 downto 0);
            D_OUT : out std_logic_vector(7 downto 0);
              CLK : in std_logic;
      DR_IN,DL_IN : in std_logic); -- right and left side inputs
end univ_sr;

architecture my_sr of univ_sr is
   signal tmp_D : std_logic_vector(7 downto 0);
begin
   process (CLK,SEL,DR_IN,DL_IN,P_LOAD)
   begin
      if (rising_edge(CLK)) then

         case SEL is
            -- do nothing (don't change state) --------------
            when "00" =>  tmp_D <= tmp_D;

            -- parallel load --------------------------------
            when "01" =>  tmp_D <= P_LOAD;

            -- shift right ----------------------------------
            when "10" =>  tmp_D <= DL_IN & tmp_D(7 downto 1);

            -- shift left -----------------------------------
            when "11" =>  tmp_D <= tmp_D(6 downto 0) & DR_IN;

            -- default case ---------------------------------
            when others => tmp_D <= "00000000";
         end case;

      end if;
   end process;

   D_OUT <= tmp_D;

end my_sr;
```

Figure 3-36: VHDL model for a universal shift register.

Figure 3-36 definitely shows some new and cool stuff in the VHDL. Namely, there are a couple of instances of the "&" operator. This is the concatenation operator in VHDL; it concatenates two signals (or parts of two signals) together. The other thing to notice is that there are two new inputs to the universal shift register. These are the DR_IN and DL_IN inputs. The DR_IN input provides new data into the right side of the shift register

during a shift-left operation while the DL_IN input provides data into the left side of the shift register during shift-right operations.

Figure 3-37 shows a black box diagram for the model of Figure 3-36. In the end, the circuit has so many functions that we simply don't need to use all of them in our problem solution. Therefore, the next step in this process is to figure out which ones we actually do need. The solution that follows is one approach to the final circuit and solution; it is not the only way to do it. The point here is to think about how you're going to solve the problem with the devices you've decided to work with. In other words, map out an approach in your head that you can use as a road map to implementing the solution. In other words, you should be asking yourself this: "How can I use the shift register to make it implement the required functionality?" In this context, the word "use" is should more appropriately be "control" since our solution will be FSM-based.

What we're trying to do now is to define the data inputs, outputs, and control and status signals that we will use in our design. An overview of these signals and the justification for their use follows; Figure 3-38(a) shows the final circuit in block diagram form.

- **D_OUT**: these are the outputs of the SR and drive the LEDs. The outputs form the "data output" outputs in Figure 3-33.

- **D_LOAD**: these inputs will put the display in to a known state. The parallel load feature places the shift register into a known state. The problem description states that the shifting will start from the "00000001" state which represents only the far right LED lit. From this point, the output will start shifting left. These inputs form the "data input" inputs of Figure 3-33.

- **DR_IN, DR_OUT**: In the solution that we'll be implementing, we won't need to utilize these inputs. Therefore, you can tie them either high or low. Other solutions this problem may utilize these inputs. Additionally, these inputs would also fall into the category of "data inputs" inputs from Figure 3-33.

- **SEL**: these inputs control the operation of the shift register. We'll be needing each of the shift register functions in this design since we have requirements to hold, shift left and right, and parallel load. These inputs form the "control inputs" in Figure 3-33.

- **?? Control Inputs ??:** The big question here is what inputs do we need to know about the status of the universal shift register? The only true outputs of the "datapath" (the shift register in our case) are the basic shift register outputs so we know we'll need to look at some of these. Do we want to look at all of them? The answer comes from thinking back to the original problem. The only thing we need to know is when the lit LED has arrived at the left-most and right-most LED. This means that we only need the D_OUT(7) (the left-most LED) and the D_OUT(0) (the right-most LED) as status inputs to the FSM.

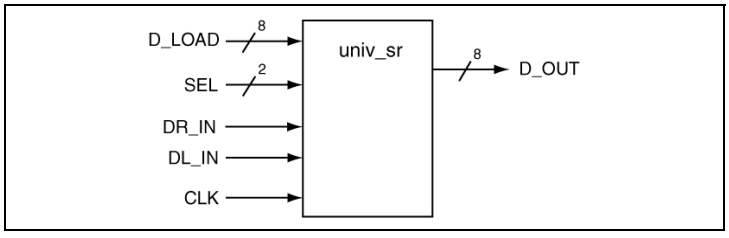
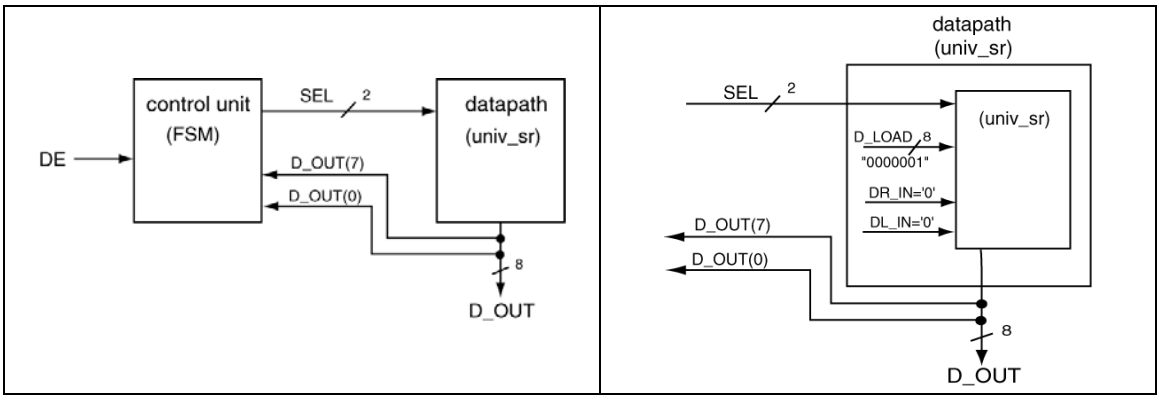


Figure 3-37: A black box diagram of the universal shift register.

Figure 3-38(a) shows the final circuit while Figure 3-38(b) shows a highlight of the interior of the datapath box. The thing that Figure 3-38(b) is actually trying to show is that we hardwire some of the inputs on the universal shift register to preset values. Because of this, Figure 3-38(a) does not show these signals.



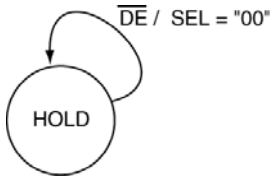
(a) (b)

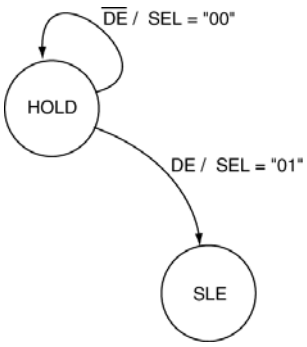
Figure 3-38: The final circuit (a) and a highlight of the datapath (b).

3.7.4 Generating the State Diagram

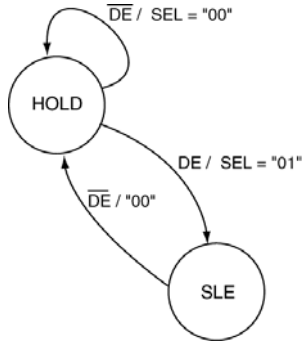
The following set of figures show the development of a state diagram that you could use to implement this solution. We explain each of the individual steps in the diagrams that follow. This is my view of how the state diagram progresses and is not necessarily a description that you'll be able to immediately relate to and understand. Therefore, it is imperative that you convince yourself of the logic behind each of the following steps. If you are able to understand the approach I took to the problem, you'll be able to generate your own approach (which will undoubtedly be better than mine).

The first step is to think about the problem in terms of what is really happening. How many states are there going to be in the final diagram? Before you start designing any FSM from a word description, you probably won't know the answer to this question. The approach you should take on this type of a problem is to decompose the problem in to individual functions: how many different actions will the final circuit take? If you think about it for a second, you can reason that there are only three distinct actions that are required of the final circuit: a hold function, a shift-left function, and a shift-right function. Most likely, these functions will translate to three states. This is a guess; let's generate the state diagram and see what happens.

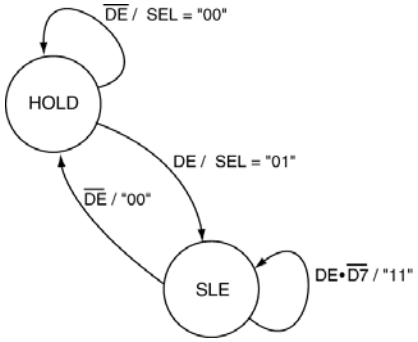
	**STEP 1:** This is the starting point. The "hold" state is probably the simplest state in the FSM, which indicates it's a good starting point. The reality is that it does not matter where you start when generating a state diagram: the final product (if you do it right and it works) should always be the same. Note that the "hold" mnemonic has been used to aid in the design and understanding of this problem.
	**STEP 2:** This is the first transition. It is probably the most straightforward transition in the ensuing state diagram. This transition is characterized by "keep doing nothing" which is in this case means to no change the display. The condition for this transition are for the DE signal to be in the '0' state. In this transition, you need to make sure the USR is not active which we do by setting the SEL signals to "00". This causes a hold condition to happen in the USR, which means the current USR outputs do not change on the active clock edge.



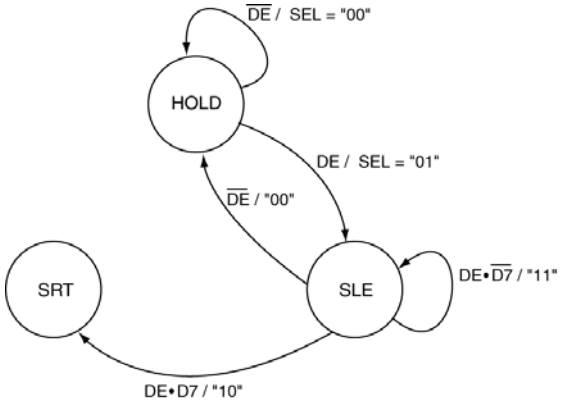
STEP 3: The next transition is the starting condition for the shifting mode. This transition is made when the DE signal is a '1'. This signal is asynchronous for this problem. When the DE signal transitions to a '1', a "01" is placed on the SEL lines which causes a parallel load on SR. Note that the SEL lines have been made into Mealy-type outputs so when the DE signal transitions from '0' to '1', the SEL lines change from "00" to "01" (hold to parallel load on the shift register). In order to start the shifting sequence on the SR, the signals that are parallel loaded are hard-coded to "00000001". This has the effect of lighting the right-most LED on the display. The new state that appears in the state diagram contains the mnemonic "SLE" which stands for shift left. Remember, when the lit LED is in the far right position, the only place it can go from there is toward the left, hence the SLE label.



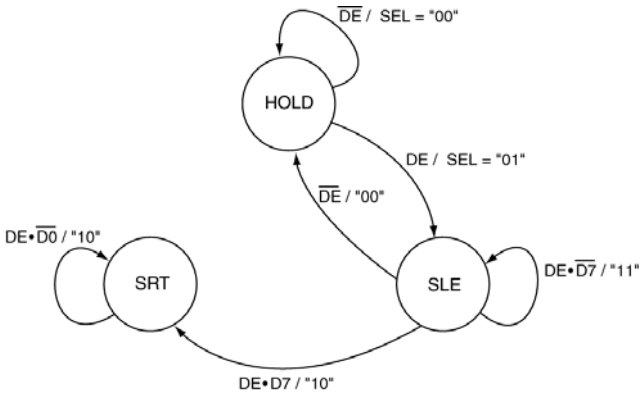
STEP 4: The next transition handles the case there the DE input returns to '0' during the shift left sequence. In this case, the current state of the LED display is preserved because the SEL input is placed in the hold condition (SEL = "00"). Note that I've stopped writing SEL in the state diagram. We now know that the two numbers on the right side of the forward slash are the two SEL outputs.



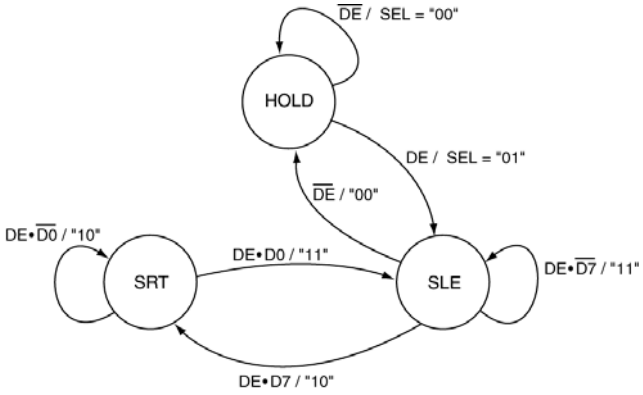
STEP 5: This step involves setting up the conditions that are required to keep the SR shifting left. The conditions include keeping DE in the '1' state and making sure that D7 is also in the '0' state. In this case, D7 represents the left-most LED. The thought there is that I want to keep shifting left so long as the left-most LED is not lit (the left-most LED will be lit when D7 = '1'). These two conditions both have to be "true" in order to keep shifting left. As for the SEL outputs, I simply need to keep them in the "11" state which forces the SR to shift-left on each clock edge.



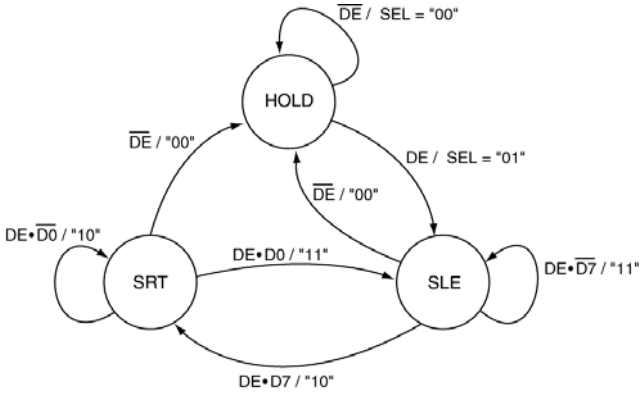
STEP 6: The next logical step is to account for the case where the SR needs to start shifting right. I know I've reached this condition when the D7 signal (the left-most LED) is a '1'. These conditions include having DE = '1' and D7 = '1'. When these conditions are present, I need to move from the shifting-left state to the shifting-right state. The shifting-right state is the new state in the state diagram. The shifting-right state is represented by the "SRT" mnemonic. The SEL outputs are set to "10" which represents the shift-right state for the shift register.



STEP 7: This step involves setting up the conditions that are required to keep the SR shifting right. The conditions include keeping DE in the '1' state and making sure that D0 is also in the '0' state. In this case, D0 represents the right-most LED. I want to keep shifting right so long as the left-most LED is not lit (the right-most LED will be lit when D0 = '1'). These two conditions both need to be "true" in order to keep shifting fight. As for the SEL outputs, I simply need to keep them in the "10" state which forces the SR to shift right on each active clock edge.



STEP 8: The next step is to account for the case where the shift register needs to start shifting left. I've reached this condition when the D0 signal (the left-most LED) is a '1'. The conditions for this transition includes having DE = '1' and D0 = '1'. When these conditions are present, I need to transition from the shifting-right state to the shifting-left state. The SEL outputs are set to "11" which forces a shift left state on the shift register.



STEP 9: The final transition in the state diagram account for the condition where the hold state is entered from the shift-right (SRT) state. This transition occurs when the DE input is placed at '0'. The resulting hold condition is then accomplished by setting the SEL outputs to the "00" state. Happiness now exists. From here on out, all you need to do is choose a method to implement the state diagram.

3.7.5 Some Specifics of the FSM Timing

There are some peculiarities associated with the actual timing of this circuit. A brief discussion of these areas hopefully will enhance your interest and enjoyment of FSMs by increasing your basic understanding of timing matters associated with state diagrams. Figure 3-39 shows the first interesting timing consideration.

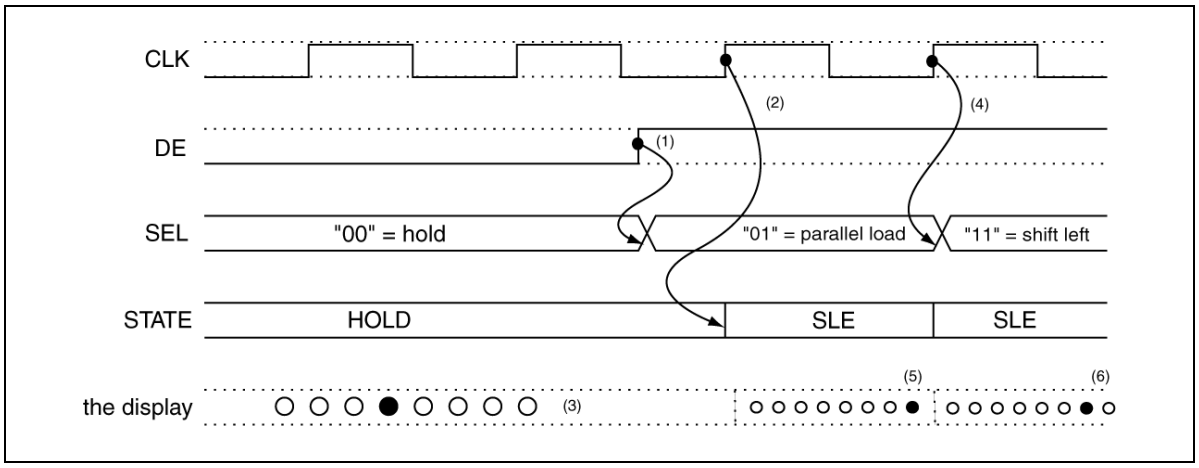


Figure 3-39: A typical starting sequence.

Figure 3-39 shows the timing sequence associated with the transition from the HOLD state to the SLE. When the DE signal is asserted, it causes the SEL outputs to immediately change as is indicated by (1) in Figure 3-39. This is because the SEL outputs are Mealy-type and are a function of the DE inputs. The change in the DE and SEL signals has no immediate effect on the datapath circuitry because we synchronize both the state transition and the display changes to the active clock edge (which we assume to be the rising-edge in this case). The note (2) in Figure 3-39 indicates this condition.

The changing of the output data (the display) occurs on the clock edge is shown between notes (3) and (5). For the (3) note, the light pattern shown in Figure 3-39 is arbitrary for this example timing diagram. This indicates that the device was previously returned to the HOLD state when the LED shown in (3) was lit (a dark circle indicates a lit LED). The change that occurs from note (3) to (5) in the display line is based on the parallel load as which is caused by the SEL lines being set to “01”.

The parallel load is synchronous with the clock edge. The crux of this matter is that both the state (from the control unit) and the display (from the datapath) change simultaneously. The clock edge shown in note (4) causes a left shift. Note that the “11” on the select outputs are associated with the outputs of the SLE state and not the HOLD state as was the “01” output. The shift left output causes the lit LED to shift from the position shown in note (5) to the position shown in note (6). Is this cool or what? Be sure to compare and contrast this timing diagram with the final state diagram shown on a previous page to help you understand timing issues such as these. Mealy-type outputs are definitely tricky. Remember that the output associated with a state transition arrow belong to the state it is exiting from.

Figure 3-40 shows another interesting timing diagram. In this case, the timing diagram shows the transition from a shift-left to a shift-right. The interesting thing to note in this diagram is that when the left-most LED is lit, it causes the SEL inputs to be set to the shift-right value. In this way, on the next clock edge, the LED will officially shift right even though the FSM is still in the SLE (shift-left) bubble. This is another one of those peculiar characteristics associated with a Mealy-type output. I encourage you to once again examine the final state diagram for this FSM and convince yourself that the timing diagram shown in Figure 3-40 is true and correct. But most importantly, be sure you completely understand the relationship between the timing diagram, the state diagram, and the actual functioning of the circuit.

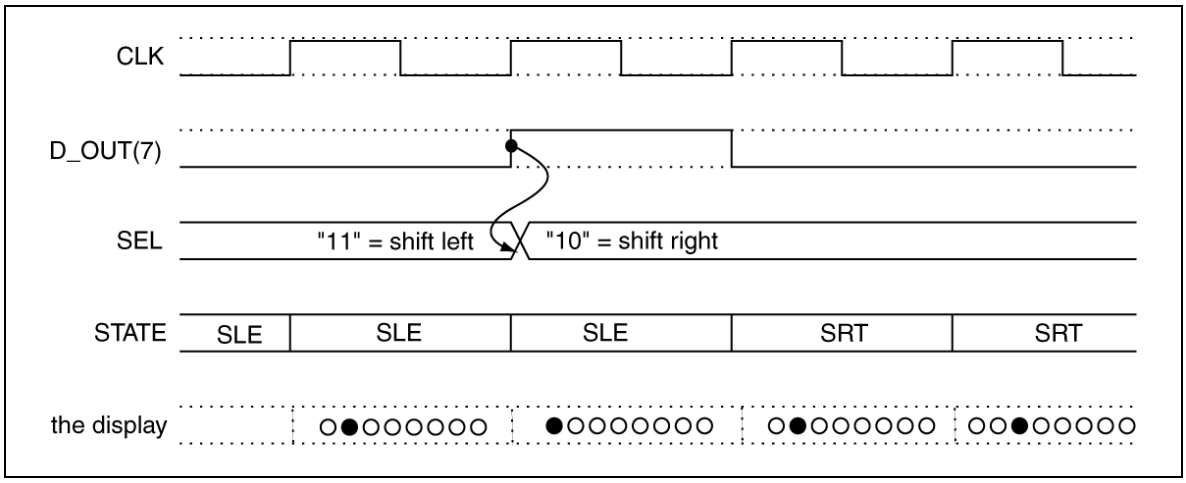


Figure 3-40: The timing associated with a shift-left to shift-right transition.

3.8 Chapter Summary

- All digital circuits can be categorized as being either a combinatorial or a sequential circuit. Combinatorial circuits do not have memory and their outputs are a simple function of their inputs. Sequential circuits have the ability to store bit, thus making their outputs a function of the sequence of inputs.
- All digital circuits, including the most complex digital circuits, comprise of a set of basic digital design modules. These modules include both combinatorial and sequential circuits.
- The main combinatorial digital building block circuits are built from simple logic gates. These circuits include the following:
- Half and full adders: circuits capable of adding two 1-bit values
- Ripple Carry Adders: circuits comprised of half and full adders chained together to form “n-bit” adders.
- Multiplexors: circuits used as signal selection circuits
- Decoder: circuits used to establish a given relationship between the circuit inputs and output.
- Comparators: circuits that compare two values and provides information regarding the relationship between the two input values; well-known to be made with EXOR-type gates.
- Parity Circuits: circuits used to establish or check the parity of circuits; well-known to be made with EXOR-type gates.
- Flip-flops: one-bit synchronous storage elements
- Finite State Machines (FSMs): circuits that have sequential and combinatorial elements typically used as controllers for other digital circuits.
- VHDL is a language typically used to model digital circuits. VHDL models can also be used to synthesize actual circuits. VHDL is also useful to test other VHDL models with the use of VHDL “testbenches”.

3.9 Chapter Exercises

1) In your own words, briefly describe what we mean by the term “digital bag of tricks”.

2) Briefly describe the main differences between a combinatorial and sequential circuit.

3) In your own words, describe the relation between memory of a sequential circuit and the notion that the outputs are a function of the “sequence” of inputs to the circuit.

4) Briefly describe what characteristic gives a circuit the ability to store bits.

5) Briefly describe the difference between a half adder and a full adder.

6) Briefly describe how a “ripple carry adder” was given such a name.

7) At any given time, how many AND gates in a multiplexor are not dead? Briefly explain your answer.

8) In your own words, briefly describe the difference between a generic decoder and a standard decoder.

9) What basic component do parity generators, parity checker, and comparators all share.

10) Briefly describe the difference between a flip-flop and a latch.

11) Briefly describe the differences between a Mealy and Moore-type FSM.

12) Briefly describe what characteristic of VHDL code generates memory in VHDL models.

PART TWO: Hardware Foundations of Computer Design

4 Basic Registers

4.1 Introduction

I have no trouble stating that most commonly used circuit in digital design is the "register". We've already used the term quite often in this text, particularly regarding finite state machines (FSMs). Recall that a main component of FSM was the storage associated with the state variables. Although I did my best not to use the word register, there were several instances when I used the term "state registers" to refer to the circuit elements storing the state variables.

The concept of registers is not complicated and you've been dealing with the basic register concepts for many chapters at this point. This chapter describes the notion of registers and their many various flavors and incarnations. Most of the description appearing in this chapter is at a higher-level as the low-level details are somewhat cumbersome and not overly useful. All forms of registers are massively useful in digital design. The next chapter examines some specific instances of common and useful registers.

Main Chapter Topics

- **SIMPLE REGISTERS AND REGISTERS "WITH FEATURES":** This chapter defines and describes basic including registers with extended features that make them more useful in digital circuits.
- **TRI-STATE DEVICES AND TRI-STATE REGISTERS:** This chapter describes tri-state devices and their use in tri-state registers and associated circuitry.
- **BI-DIRECTIONAL REGISTERS:** This chapter briefly describes the notion of bi-directional registers and their relation to tri-state registers.

Why This Chapter is Important

This chapter is important because registers and their simple variations are extremely useful and thus often found in just about all meaningful digital designs.

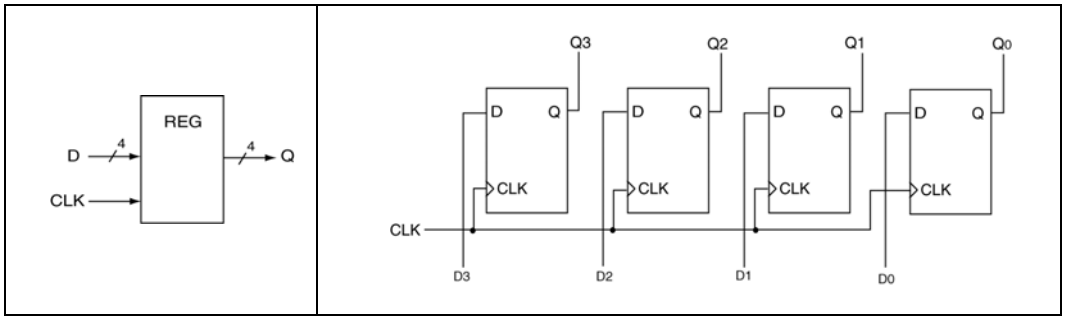
4.2 Registers: The Most Common Digital Circuit Ever?

Stated as simply as possible, a register is nothing more than a multi-bit flip-flop. Flip-flops are single bit storage elements while registers multi-bit storage elements modeled as a given number of flip-flops sharing the same clock signal. The good news is that we only use D flip-flops to model registers, which simplifies their understanding and representation. Moreover, VHDL models of registers are similar to flip-flop models and only differ in the width of the "data" inputs and outputs. In addition, being that registers such as these have such simple descriptions, we'll refer to this flavor of registers as "simple registers". For now and evermore, when we say, "register", we typically mean "simple register"; this works well as the more specialized registers have their own names. A future chapter describes more advanced and functional registers.

Jumping right into it, Figure 4-1 shows four D flip-flops assembled such that they act as a register. In particular, Figure 4-1(a) shows the block diagram for a 4-bit register and Figure 4-1(b) shows the underlying circuit. Here are a few things to note about Figure 4-1:

- The block diagram in Figure 4-1(a) shows a clock signal but also assumes other characteristics. Since we model the register with D flip-flops, there must be an active clock edge not shown in Figure 4-1(a). Unless otherwise stated, registers are generally active on the rising-edge of the clock, which is what Figure 4-1(b) shows. You generally don't see registers in the Figure 4-1(b) form though; the model in Figure 4-1(a) is a better representation in that it is more highly abstracted.

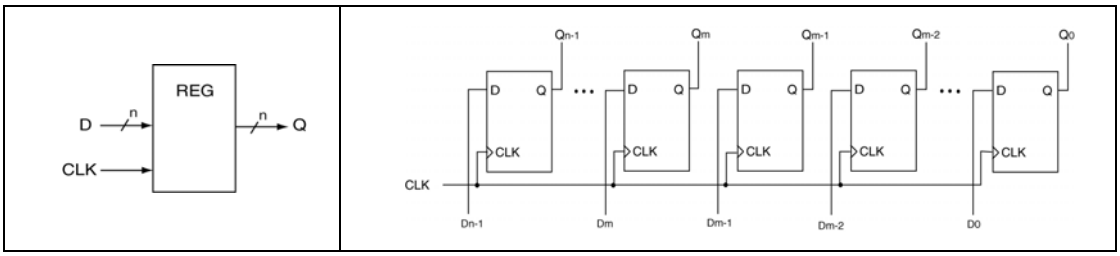
- Figure 4-1(b) shows that each flip-flop in the register shares the same clock. The result is that all the flip-flops latch their data simultaneously. We'll demonstrate this later in a timing diagram.



(a) **(b)**

Figure 4-1: A block diagram for a 4-bit register (a), and the lower-level implementation details of a 4-bit register (b).

Figure 4-2(a) shows the block diagram for a generic n-bit register; Figure 4-2(b) shows the underlying details. The main point behind Figure 4-2 is to show the notion that registers are simple to model and it takes about zero effort to model registers of any width. The only thing about registers that may be somewhat tricky is the notion that an n-bit register is generally modeled using n signals: the least significant bit (LSB) has an index of "0" while the most significant bit (MSB) has an index of "n-1". Get used to it.



(a) **(b)**

Figure 4-2: A general case of an n-bit register; the block diagram (a), and a model for the underlying circuit (b).

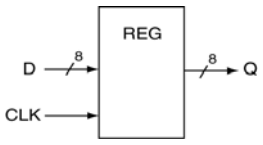
Probably the happiest way to describe the operation of a register is with the associated VHDL model. Figure 4-3 shows a VHDL model for an 8-bit register. From this model, you can see that the register is in fact active on the rising clock edge. There is not a lot to say about the VHDL model shown in Figure 4-3 as this model should appear familiar since it resembles a simple model for a D flip-flop. In truth, D flip-flops are naturally easy to model in VHDL mainly because the main goal in design the language was to be able to model popular digital circuits such as registers without expending too much effort.

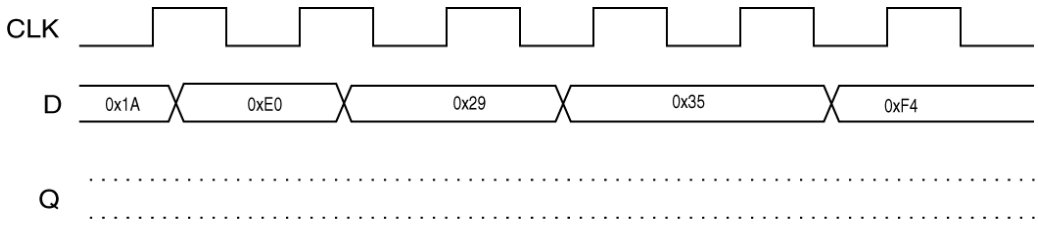
```
-------------------------------------------------------------
-- VHDL model of 8-bit register
-------------------------------------------------------------
entity reg_8b is
    Port (   D : in std_logic_vector(7 downto 0);
           CLK : in std_logic;
             Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin

   process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process;

end my_reg_8b;
```

Figure 4-3: The VHDL model for a simple 8-bit register.

Example 4-1: A Simple Register

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered; ignore all propagation delay issues.





Solution: From the problem description, we know the block diagram represents an 8-bit register that is active on the rising clock edge. This means that we need to examine only the portions of the timing diagram aligned to the rising edge of the clock. At these times, the data on the input of the register transfers to the output of the register. Figure 4-4 shows the result for this example. The final solution is straightforward but has a few items worth noting.

- The solution adds dotted vertical lines on the rising clock edges. This is something you should always consider doing when working with timing diagrams. Timing diagrams can quickly become quite complexicated so drawing dotted lines such as these is a good first step in solving these problems.

- The problem did not provide an initial value for the contents of register. Because of this, the first time-slot on the "Q' line contains question marks. In other instances, the problem may either state an initial value or provide some type of signal that places the register into a known state. We'll see such an asynchronous "reset" signal in a later example problem to deal with this issue.

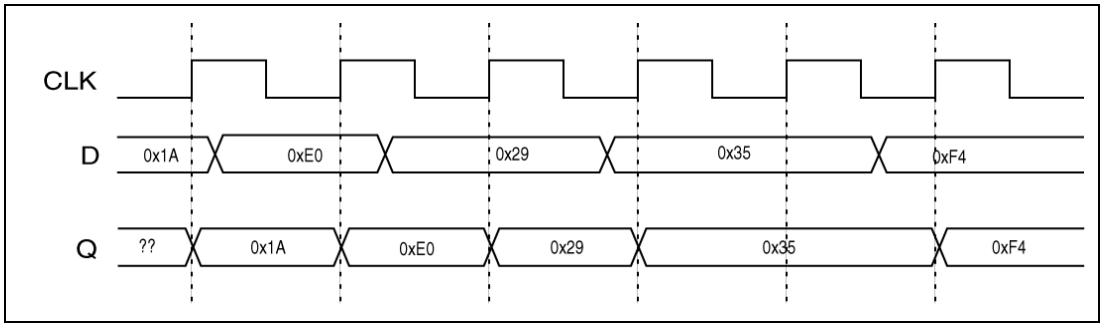
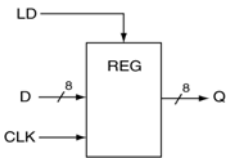


Figure 4-4: The solution for Example 4-1.

In real digital circuits, you rarely see registers as simple as the register in Figure 4-3. These registers don't have enough "control" to make them useful. The issue is that at every active clock edge, the register latches the input data. Real registers generally contain control signals that direct when the register latches data.

Without too much explanation, Figure 4-5 shows a register containing a signal that controls if the register latches the input data. Control signals for such registers are typically associated with the word "load" (and the acronym "LD"; registers typically "load" the input data into the register. As a result, we use "LD" as a signal name for the control signal in the register in Figure 4-5. This signal allows each of the single-bit storage elements in the register to latch their associated bits. In particular, Figure 4-5(a) shows a block diagram for the register with a control signal while Figure 4-5(b) shows the associated VHDL model. The following example shows the operation of this register.



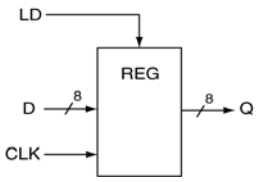

```
-------------------------------------------------------------
-- VHDL model of 8-bit register with parallel load
-------------------------------------------------------------
entity reg_8b is
    Port (      D : in std_logic_vector(7 downto 0);
           CLK,LD : in std_logic;
                Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin

   process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         if (LD = '1') then
            Q <= D;
         end if;
      end if;
   end process;

end my_reg_8b;
```

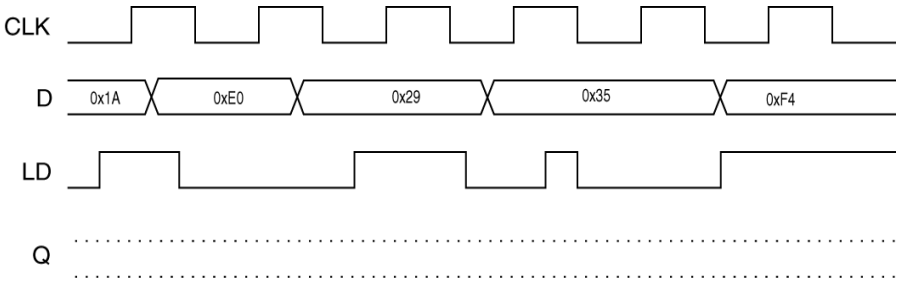
(a) **(b)**

Figure 4-5: An 8-bit register with a parallel load signal (a) and an associated VHDL model (b).

Example 4-2: A Register with Load Control

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered; ignore all propagation delay issues.





Solution: This problem is similar to the previous problem but now we need to keep track of the "LD" signal. In the previous problem, we only needed to examine the times when the rising edges occurred. In this problem, we need to examine the times where the both the rising edge occurs and where the LD signal is asserted. Note that because the LD signal on the register does not have a bubble, the load signal is active high. Figure 4-6 shows the solution for this example; some interesting things to note surely follow as well.

- We explicitly mark the rising edges with vertical dotted lines in order to avoid confusing ourselves.

- The problem statement does not provide an initial value for the register so we must mark it as unknown. The question marks work well for this dilemma.

- The LD signal is "level sensitive" which essentially means that it is not edge sensitive. This means that we are only concerned with the register loading when the LD signal is asserted and not only on the rising edge associated with the signal. In Figure 4-6 at the time marked with the circled "1", the LD signal asserts and then de-asserts shortly thereafter. This small pulse has no effect on the register because there was not a rising clock edge present when the LD signal was asserted.

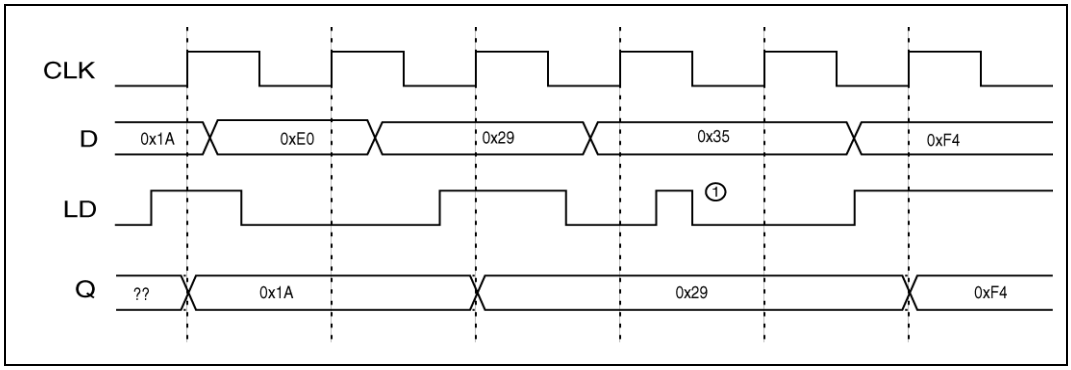
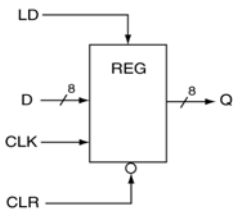


Figure 4-6: The solution for Example 4-2.

Registers can have other control options also. The notion here is that if you need to use a register in your circuit, you choose the register with the smallest feature set but still allows you to get your job done. Extra features in circuits require extra hardware; extra hardware takes up space and requires extra power to operate. If you're designing your own registers, such as in a VHDL application, you have the ability to design any feature into the device that your circuit requires. The next example we'll look at has one more added feature; after that, we'll stop talking about registers.

Figure 4-7(a) shows a register that has both a load and a clear input. Because this is a register, everyone generally assumes that the load signal is synchronous. The clear signal is usually asynchronous, but not always. The moral of this circuit is that you should make sure you know everything there is to know about the circuit; making assumptions is usually problematic. What saves us on Figure 4-7(a) is that Figure 4-7(b) provides the VHDL model for the diagram. If you read the associated VHDL model, you can see that the LD signal is synchronous while the CLR signal is asynchronous. An example problem shows how this circuit operates.



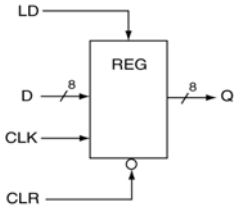

```
-------------------------------------------------------------
-- VHDL model of 8-bit register with load and clear
-------------------------------------------------------------
entity reg_8b is
    Port (          D : in std_logic_vector(7 downto 0);
           CLK,LD,CLR : in std_logic;
                    Q : out std_logic_vector(7 downto 0));
end reg_8b;

architecture my_reg_8b of reg_8b is
begin
   process (D,CLK)
   begin
      if (CLR = '0') then   -- asynchronous clear signal
         Q <= (others => '0');
     elsif (rising_edge(CLK)) (then
         if (LD = '1') then
            Q <= D;
         end if;
      end if;
   end process;

end my_reg_8b;
```

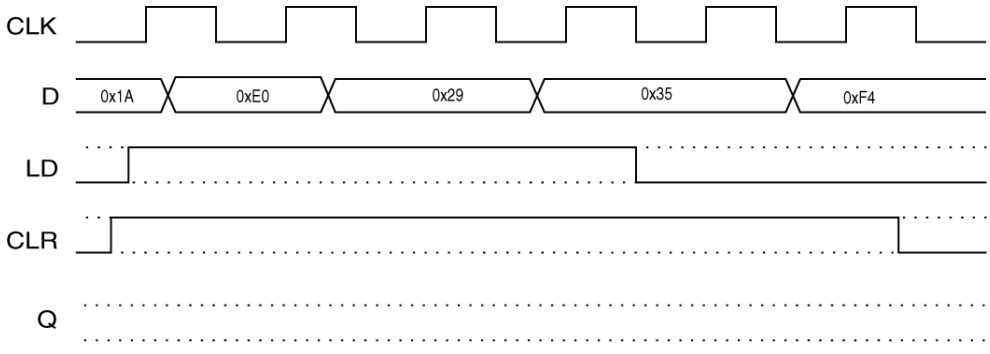
(a) **(b)**

Figure 4-7: An 8-bit register with a synchronous parallel load input and an asynchronous clear input (a), and a VHDL model for the underlying circuit (b).

Example 4-3: A Register with Synchronous and Asynchronous Control

Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues. The LD input is a synchronous parallel load input while the CLR signal is an asynchronous active low signal that clears the register when asserted.





Solution: Although this solution is rather straightforward, it provides a few new tidbits of information regarding the operation of registers.

- Unlike the previous examples, the asserted CLR signal at the beginning of the timing diagram makes the value stored in register a known value. Because of the asserted CLR, the register clears all the internal storage elements as the timing diagram indicates.

- Though you can't tell from the first instance of the asserted clear signal, the second instance shows that the CLR signal is actually asynchronous. We know this because the clearing of the output register occurs shortly after the CLR signals asserts near the end of the timing diagram.

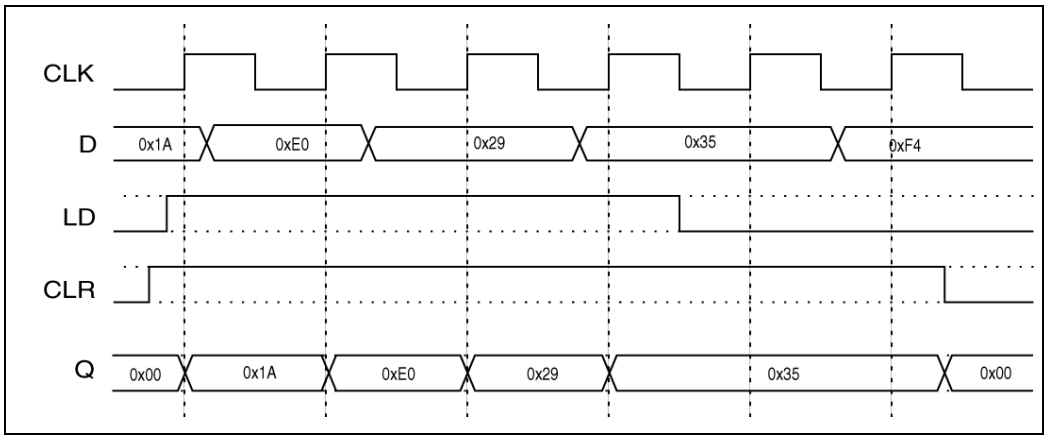


Figure 4-8: The final solution to Example 4-3.

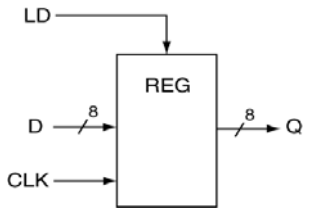
A few final words on registers… Registers come in many different flavors; this section presented only a few of them. As you'll find out later, a few other common devices out there in digital-land are essentially highly specialized registers. When someone mentions a "register", they are typically referring to the register we describe in this section. The other types of common registers have their own special names ("shift registers" and "counters"); the next chapter describes these registers.

This section has modeled basic registers in VHDL using both an entity and architecture. While this is a viable approach to modeling registers, it is not always the "best" approach. Registers are relatively easy to model and thus do not require a significant amount of code. Thus, you may not want to include an entity with your model, which requires that you use structural modeling to use your register in a circuit.

A simpler approach is to model registers based on signals. By signals, we mean actual VHDL signals, which we list in the declarative region of the VHDL architecture. If we model registers this way, we don't need to include entity descriptions.

Example 4-4: A Register Model without an Entity

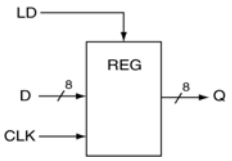
Provide the code that you could place in a VHDL architecture that models the register on the right. Assume that D, LD, and Q are internal signals to the architecture.



Solution: Figure 4-9 shows the solution to Example 4-4. Although this solution is rather straightforward, it provides a few new tidbits of information regarding the proper implementation and labeling of the register.

- The storage element associated with the register is the"r_Q" signal. In actuality, "r_Q" is truly a signal, but we're using it to induce storage using the same incompletely specified if statement as we do with other registers modeled in this chapter. The key here is to use the "r_" prefix if you choose to model registers in this way. If you did not use this prefix, your code would be hard for another human to read. Using the "r_" prefix states that although "r_Q" is declared as a signal, this model is using it as a register. Note that the other signals retain the "s_" prefix as they are truly signals and not registers.

- The code appears to contain only one process. You can officially consider this code to be a fragment; this solution lists none of the other useful stuff that you would typically find in this solution



(a)

```
-------------------------------------------------------------
-- VHDL fragment for an internal 8-bit register with
-- parallel load
-------------------------------------------------------------

architecture my_ckt of my_ckt is
   signal s_LD std_logic;
   signal r_Q std_logic_vector(7 downto 0);
   signal s_D std_logic_vector(7 downto 0);
begin

   process (D,CLK)
   begin
      if (rising_edge(CLK)) then
         if (s_LD = '1') then
            r_Q <= s_D;
         end if;
      end if;
   end process;
```

(b)

Figure 4-9: An 8-bit register with a parallel load signal (a) and the VHDL code that models that register not using an associated entity (b).

4.3 Tri-State Registers

Although the underlying theme of digital design is the notion of binary signals, there is one other common and useful "state" in digital-land. Certain digital devices have the ability to have a third output in addition to the standard '1' and '0'. We refer to these devices as "tri-state" or "three-state" devices[1], because these devices have a third output known as the "high-impedance" state. The best way to refer to think about these devices is not to consider these devices as having a third state, but rather to think about these devices as having a magic switch that either allows the device to operate normally or kills the device altogether.

The notion of high-impedance is common in both analog and digital design. There are many ways out there to model high impedance devices, but I prefer to model them using Ohm's Law: V=IR, with V representing voltage, I representing current, and R representing resistance. For this discussion, we can consider impedance the same thing as resistance. If we rearrange Ohm's Law, we obtain the R=V/I, which states that the resistance is directly proportional to the voltage and inversely proportional to the current. In digital circuits, the voltage is generally constant so we'll only consider R and I. For the R value to be large implies

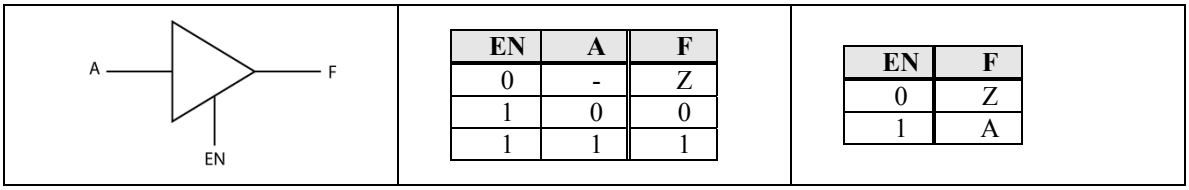
[1] The difference between "tri-state" and "three-state" is that some company trademarked one of these terms. Be careful how you use the terms; lawyers are waiting in the wings.

that the I value to be small. When I is small means that there is little current flowing in a circuit. Digital circuits require current in order to operate, so a circuit with high-impedance means the circuit has low current, which implies the circuit is dead. Yet another way to model high-impedance is as a switch that turns off the current to a circuit; an open switch is the same as an open circuit or broken circuit, which implies the circuit is dead.

There are many great reasons out there for you to kill your digital circuit. The two major reasons in digital design are to 1) save power, and 2) give your circuit the ability to share resources. The notion of sharing resources is massively important and useful; we'll exercise this notion in a later chapter in the context of actual useful circuits.

Figure 6-9(a) shows the notion of a tri-state buffer. Although there are many tri-state-type devices out there, we can best explain the notion with a simple buffer. Figure 6-9(a) shows a tri-state buffer; this circuit is simply a buffer with a control input. The control input in Figure 6-9(a) is the "EN" input; this input controls whether the output of the device is in a high-impedance state or not. Another way to think of this input is as a switch that either turns on or turns off the circuit. Also note that because of the way we have drawn the circuit in Figure 6-9(a) that the control input is active high; had the "EN" input included a bubble, the control input would be active low.

Figure 6-9(b) shows a truth table that describes the operation of the tri-state device. Note that Figure 6-9(b) uses the term "Z" to represent high-impedance[2]. Figure 6-9(b) states that the buffer output is in a high-impedance state when the "EN" input is not asserted (EN='0') or the circuit is operating normally (outputs of 1's and 0's) when the "EN" input is asserted. Figure 6-9(c) shows a compressed truth table describing the circuit. Figure 6-9(b) and Figure 6-9(c) indicate that the EN (enable) input essentially enables the input to appear on the output of the device as.



EN	A	F
0	-	Z
1	0	0
1	1	1

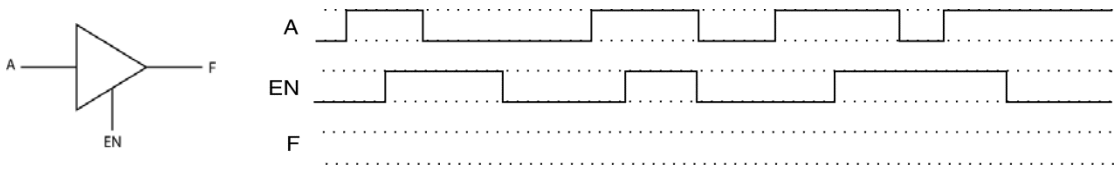
EN	F
0	Z
1	A

(a) (b) (c)

Figure 4-10: A tri-state buffer (a) and associated truth tables in full and compressed form (b) and (c).

Example 4.5: Tri-State Buffer Timing Diagram

Use the following tri-state buffer diagram to complete the following timing diagram.



Solution: Figure 4-11 shows two different forms of the solutions to Example 4.5. In reality, there are many different ways to represent high-impedance. What you'll find out in digital-land is that every datasheet and

[2] The term "Z" is how both digital and analog electronics represent high-impedance.

every simulator represents high-impedance in different ways; the two approaches in Figure 4-11 are two of the more popular approaches.

The upper F timing in Figure 4-11 uses bundle-related notation for showing when the signal is high-impedance. Note that when the EN signal is not asserted, the F output is in the high-impedance state; when the EN signal is asserted, the A input appears on the F output. The lower F timing shows the same characteristics as the upper one, but the timing diagrams shows the high-impedance output with a signal that is neither high nor low. For single signals (as opposed to bundles), the lower version of the F timing is more common. Even better, devices such as simulators that display these types of outputs typically use colors to represent the signal values such that the high-Z output is a different color than the normal digital signal.

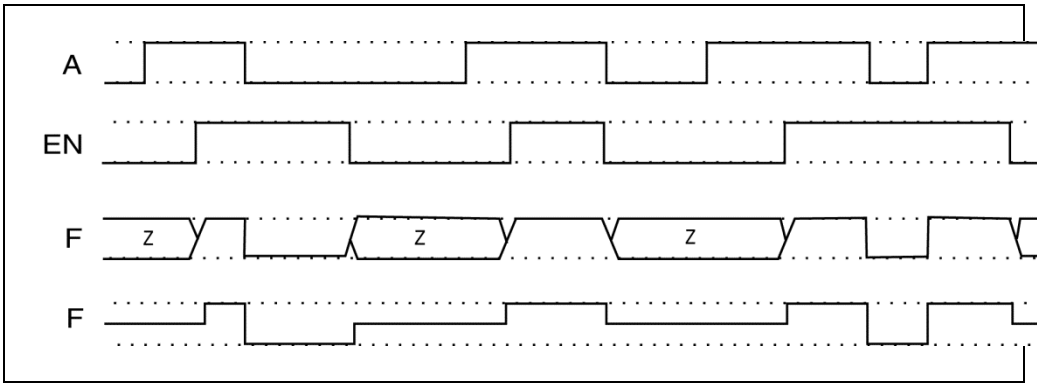


Figure 4-11: Two equivalent solutions to Example 4.5.

The notion of tri-stating applies to many digital devices. The notion of "tri-stating" is a feature of a device and thus does not come free. When your particular circuit requires a tri-state device, then you use one; otherwise, you avoid using a device with the tri-state feature to save costs. The tri-stating needs of a circuit are most often associated with circuits that share resources in an effort to reduce overall circuit size and/or costs. One particularly common tri-state device out in digital-land it the tri-state register.

The term tri-state register refers to the notion that you can place each of a register's outputs into a high-impedance state. The tri-state control input associated with a tri-state register always controls the registers output in a parallel manner. In other words, the tri-state control places either all of the circuit's output in high-impedance state when the control is asserted, or all the registers output are in a digital state when the control input is not asserted. Figure 4-12 shows a circuit diagram for a typical tri-state register. We know this device is a tri-state register because of the triangle adjacent to the OUT signal. We also know that since this is a tri-state device, the EN signal is what controls whether the output is hi-Z or a normal digital output.

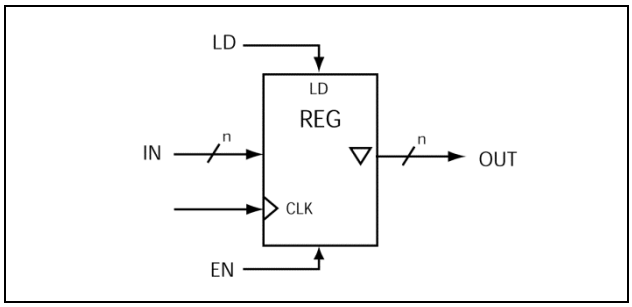
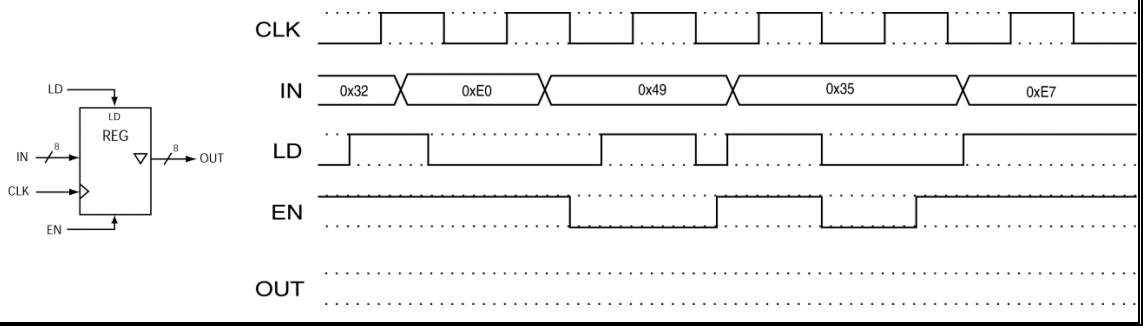


Figure 4-12: A schematic diagram of a basic tri-state register.

Example 4.6: Tri-State Register Timing Diagram

Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.



Solution: Figure 4-13 shows the solution to Example 4.6. There are several particularly import thing to note about the solution in Figure 4-13.

- Any time the EN input is not assert, the OUT signal is in its high-Z state. We arbitrarily represented the high-Z state with "ZZZ", which you should not equate with the fact that this problem is boring.
- The LD signal is effectively independent from the output. In this way, the register still loads the IN signal into the register regardless of whether the EN signal is asserted or not. This event occurs during the third rising-edge of the clock in Figure 4-13.

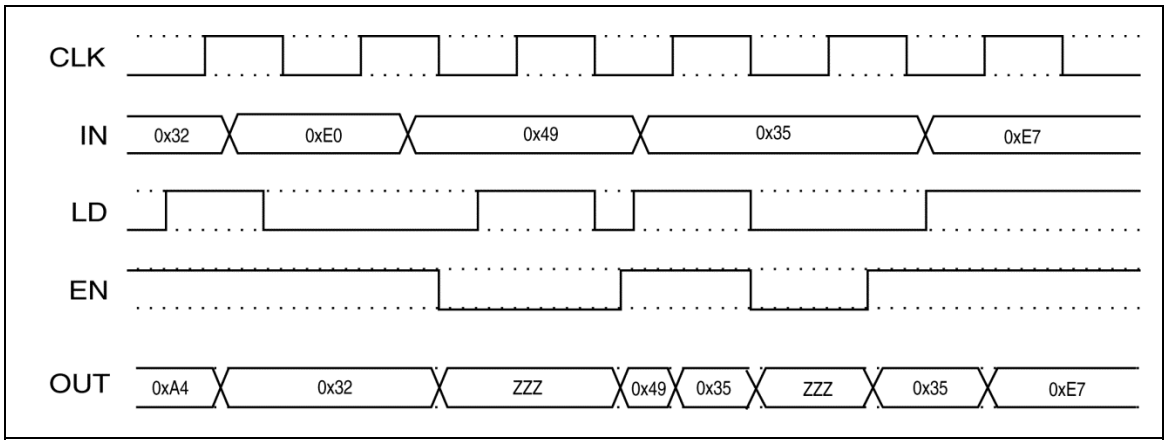


Figure 4-13: The solution to Example 4.6.

One of the reason tri-state registers exist is to save resources. This is a topic we generally save for advanced digital design, but we'll mention it here in case you never advance digitally[3]. Aside from that lame attempt at humor, the notion of using tri-state registers for resource sharing brings up a massively important point which every digital designer needs to know.

As an example of resource sharing, Figure 4-14 shows two tri-state registers in same circuit. Note in Figure 4-14 that there is a connection between the outputs of the two registers. Because these two registers are sharing the same routing resources, and because both of these devices have the ability to "drive the bus", a

[3] That would be a real shame.

potential problem exists. Enabling both registers are simultaneously creates a situation we refer to as "bus contention". Bus contention occurs when two more output devices (registers in this case) simultaneously drive their data onto the same lines bus line. Bus contention results in indeterminate circuit behavior and is thus something you should avoid. For example, if one output device drives the bus with all 1's and another device drives the bus with all 0's, what would some input device see on these lines? Who knows![4]

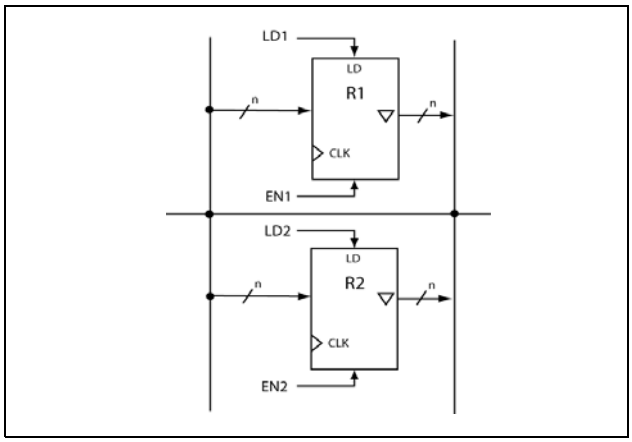


Figure 4-14: A schematic diagram of a basic tri-state register.

Working with circuits that share resources in this way certainly creates a new aspect to digital design. But all is not lost; the way to avoid bus contention is to make sure that no more than one output device is driving the bus lines at any given time. The way to "drive the bus" is to assert the enable input on the given device. Recall that when the tri-state outputs are not asserted, the device is essentially removed from the circuit as the devices outputs are providing no current to the circuit.

4.4 Bi-Directional Registers

A discussion on registers would not be complete without a description of bi-directional registers. In a continued effort to save resources, some registers use a single bundle (or bus) to route the data into and out of a register. These registers retain the required control signals including load control, tri-state control, and a clock, but the share the input and output lines. Figure 4-15(a) shows a schematic diagram of a typical bi-directional register; Figure 4-15(b) shows the same register drawn on a lower level to show some of the pertinent device implementation details. There are a few items in Figure 4-15(b) worth noting.

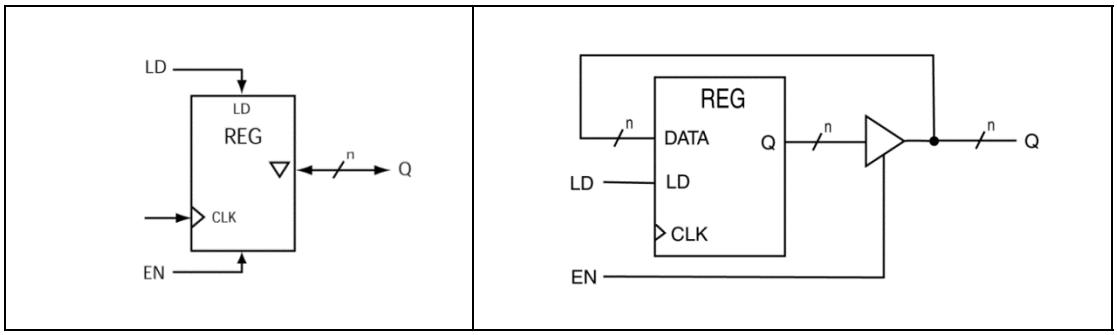
- Figure 4-15(a) represents the bi-directionality of the device with the doubly directed arrow for the Q bundle. In this way, the Q bundle can be either an input or an output depending on the EN control signal.

- The diagram uses the standard "tri-state" upside-down triangle in conjunction with the double directed arrow to officially represent the bi-directionality of the device.

- Figure 4-15(b) does not include the tri-state symbol. Figure 4-15(b) shows that you can model a bi-directional register as a standard register with a tri-state buffer on the output. The notion with this circuit is that the enable signal (EN) effectively prevents the register from driving its data to the outside world when the EN signal is not asserted. However, despite the EN signal being unasserted, the register can still latch any data that some other circuit is driving onto the data lines.

[4] But consider asking an academic administrator; they'll generate an answer for everything, particularly things they know nothing about.

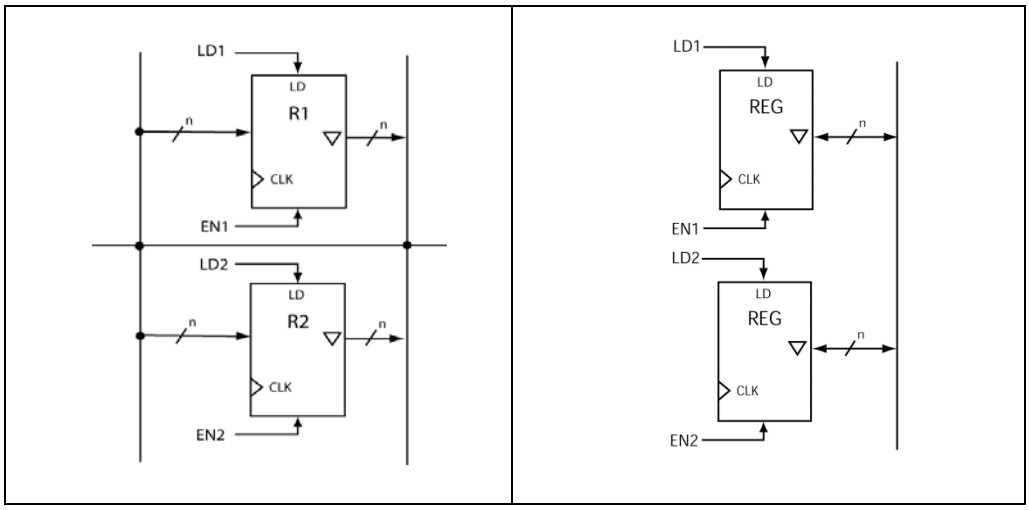
Bi-directional registers are similar to tri-state registers, but they do have a subtle difference. In tri-state registers, the enable input either drives the output or places the output into high-Z mode. In bi-directional registers, the device's enable either drives its data onto the shared resource when the enable signal is asserted, or the device is "listening" to the shared resource when the enable signal is not asserted.



(a) (b)

Figure 4-15: A circuit diagram for a bi-directional register (a), and the same bi-directional register drawn on a lower level (b).

Once again, this is a slightly advanced subject so we won't provide much more than a mention of some of the bi-directional device's functionality. We'll leave this subject with one final diagram. The notion of tri-stating and bi-directionality saves routing resources in a circuit, sometime at the cost of losing some flexibility in the circuit. Figure 4-16 shows two functionally equivalent circuits that advertise this resource sharing. Figure 4-16(a) shows a circuit with two register with tri-state outputs while Figure 4-16(b) shows a circuit with two registers with bi-directional outputs.



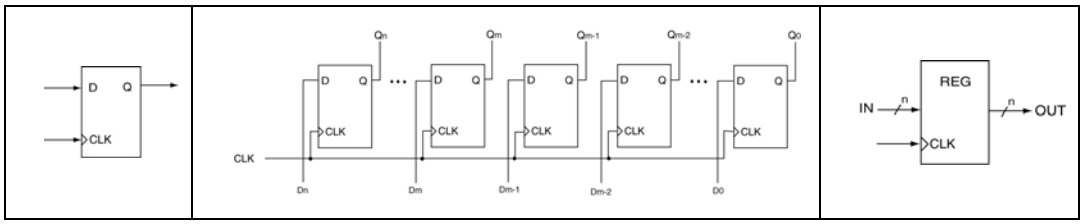
(a) (b)

Figure 4-16: Two functionally equivalent circuit models: (a) is the tri-state version of the circuit while (b) is the bi-directional version of the circuit.

4.5 Registers: The Final Comments

A register is nothing more than a set of bit storage elements that share a single clock signal. In other words, registers are a parallel configuration of signal bit storage elements; what makes them parallel is the fact that the individual storage element operations synchronize themselves to some event (usually a clock edge). D flip-flop easily model single bit storage elements; if you line up a bunch of D flip-flops together and synchronized their actions with a clock edge, you have a register.

Once you abstract all of these matters to a higher level, you'll forever more speak about *n-bit registers*. Figure 4-17 shows the progression of this abstraction. One thing to note here is that the black box diagram of a register shown in Figure 4-17 (c) includes a clock signal. The level of abstraction here sometimes continues to the point of not including the synchronizing signal (in this case, the clock) in the block diagram. In these cases, we assume the register to have a clock signal and we interpret it accordingly. Additionally, you can safely assume that all registers are edge-triggered unless told otherwise.



(a) (b) (c)

Figure 4-17: The progression from D flip-flop to register block diagram for n-bit register.

4.6 Chapter Summary

- **Registers:** A register is a sequential circuit that can be considered nothing more than a parallel combination of single-bit storage elements. These storage elements are modeled as a given number of D flip-flops that share a common clock signal and possibly other control signals typically associated with D flip-flops (such pre-set and clear signals). The register is typically used to "latch" (and thus remember) an n-bit wide set of data on the active clock edge of the device.

- **Tri-State Registers**: Tri-state registers contain tri-state buffers on the register's output. The tri-state registers effectively allow the register to either place its data onto a shared routing resource with the tri-state outputs enabled, or effectively remove itself from the circuit altogether with the tri-state outputs disabled. When the tri-state register's outputs are disabled, the circuit is "high-impedance", or "high-Z" state. An extra input signal is typically used to control the circuit's tri-state outputs. The driving notion behind tri-state register is to share, and thus save circuit routing resources, but come at the expense of overall circuit flexibility.

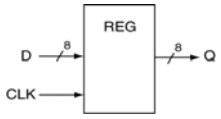
- **Bi-Directional Registers**: Bi-directional register are registers that are tri-state registers that are configured at a low-level to appear to have shared input and output lines. Bi-directional registers also represent attempts to save circuit routing resources.

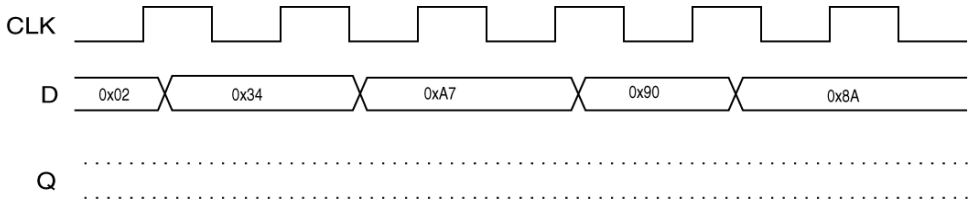
4.7 Chapter Exercises

1) Why are simple registers typically associated with D flip-flops? Would it be possible to construct a register using something such as a T or JK flip-flop? Briefly explain.

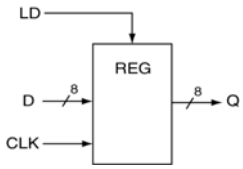
2) In what cases would there be an advantage to constructing a register (any type) using something other than a D flip-flops? Briefly explain.

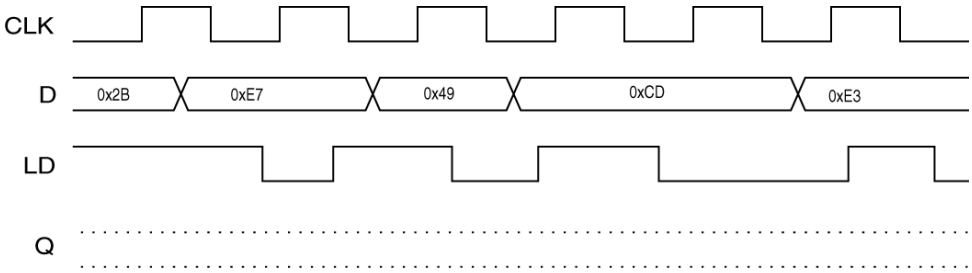
3) Using the block diagram on the right to complete the timing diagram provided below. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



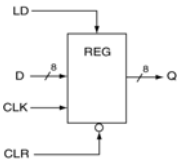


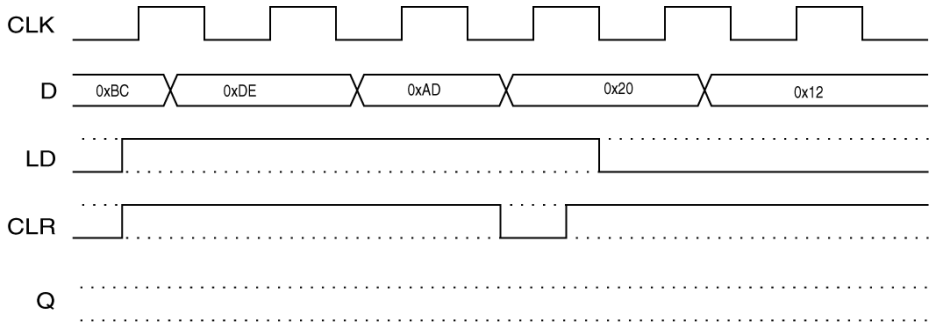
4) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



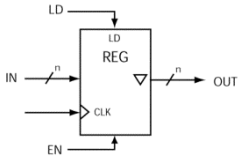


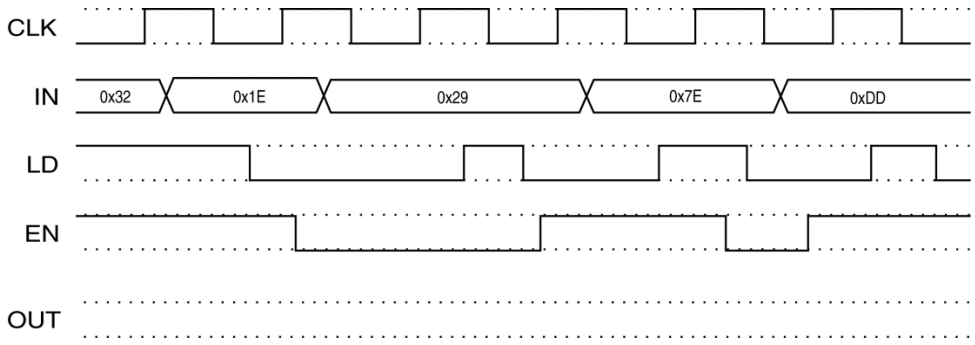
5) Using the block diagram on the right to complete the timing diagram provided below. The LD input must be asserted in order for the register to load the input signal. The CLR input is an asynchronous input that clears the register when asserted and has a higher precedence than the LD input. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



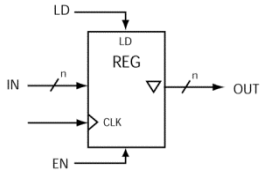


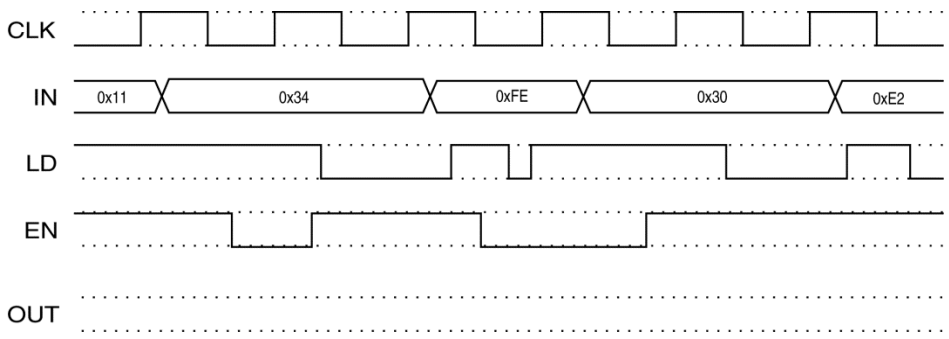
6) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xBA. Consider the register to be rising-edge triggered and ignore all propagation delay issues.



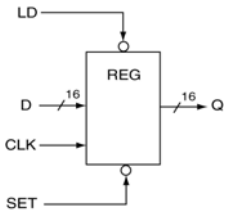


7) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xBA. Consider the register to be rising-edge triggered and ignore all propagation delay issues.

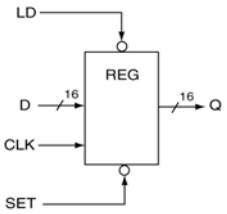




8)Provide a VHDL model that supports the black box diagram of a register on the right. The SET input is asynchronous and sets all bit storage elements in the register. The LD input loads the register on the active clock edge (falling edge triggered) and has a lower priority than the SET input.

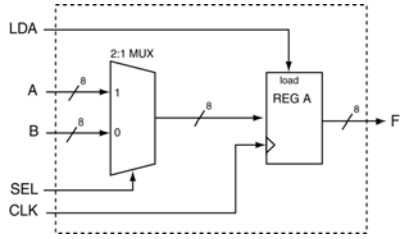


9)Using the block diagram on the right, provide a schematic diagram detailing how you would use this device to create a 32-bit register with all the same features listed on the 8-bit device.

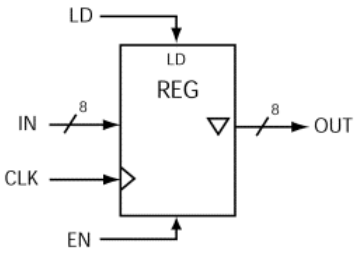


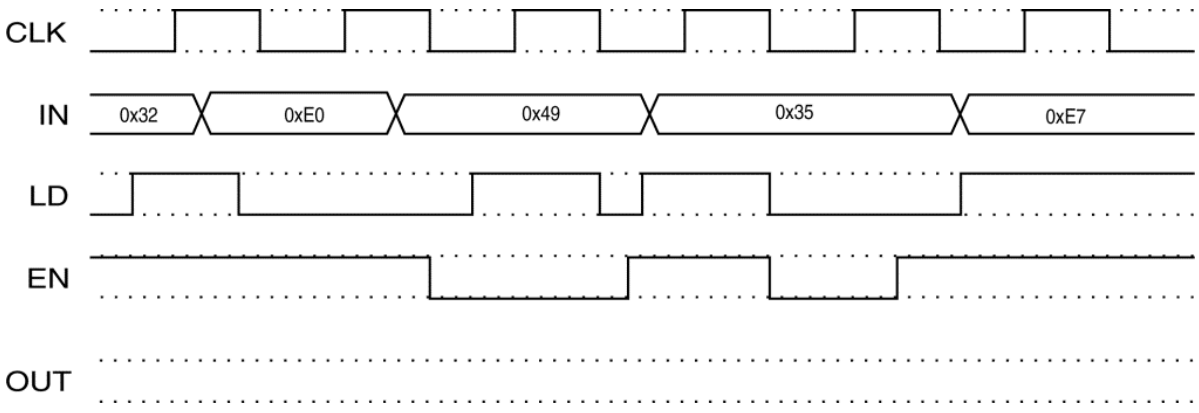
10) Write a VHDL architecture that implements the following circuit. It is not necessary to use VHDL structural modeling for your architecture; it can be done quite nicely using a combination of dataflow and behavioral modeling.

```
entity my_ckt is
   port ( CLK, SEL, LDA : in std_logic;
          A,B : in std_logic_vector (7 downto 0);
            F : out std_logic_vector (7 downto
0));
end my_ckt;
```



11) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.





5 Special Registers

5.1 Introduction

The previous chapter dealt with the notion of registers. That chapter was relatively short and not too groundbreaking, as registers are nothing more than a bunch of D flip-flops connected in parallel. The basic simplicity of registers hides the fact that they are massively useful in all forms of meaningful digital design. If you have not seen this yet, you'll for sure see it in this chapter.

Registers come in many forms: this chapter deals with two of the most common forms. As the name implies, shifter registers are nothing more than registers with special (and useful) functionality. Counters are yet another major type of specialized registers. This chapter introduces both shift registers and counters in the context of VHDL modeling. You'll surely see that even the "registers with features" are only slightly more involved than the simple register of the previous chapter.

Main Chapter Topics

- **SHIFT REGISTERS:** This chapter describes various flavors of shift registers and their basic implementations. This chapter also describes one of the main flavors of shift registers: the extremely useful barrel shifter.
- **BASIC COUNTER AND COUNTERS "WITH FEATURES":** This chapter describes various approaches to high-level VHDL behavioral modeling of counters. Previous chapters described low-level counter implementations; this chapter omits the low-level details in favor of a higher-level and thus more efficient approach.

Why This Chapter is Important

This chapter is important because shift registers and counters are extremely useful in many areas of digital design, particularly in applications requiring fast arithmetic operations. These devices are simple registers with extended features.

5.2 Shift Registers: The Most Useful Digital Circuit?

A shift register is another type of register that is surprisingly similar to simple registers. This section describes shift registers in the general case at a low level and then proceeds to describe other common types of shift registers at a higher level of abstraction. As you'll see, shift registers, and their various flavors, are massively useful devices because of their ability to perform a small but useful subset of mathematical operations in a quick and simple manner.

5.3 Basic Shift Registers

It turns out that one of the more simple circuits out there in digital-land is also one of the most useful: the friendly shift register. Shift registers, and their variants[1], are extremely useful in many digital applications primarily because they do the things they do relatively fast. Shift registers don't really do that much[2], but they do a few things really well. Basic shift register circuitry is not complicated and thus helps the noob understand the basic functioning of useful sequential circuits.

There is a simple concept behind shift register operation. In other words, we can decompose a shift register down to its most basic component, which we refer to as a shift register cell. As you would guess, this cell is nothing more than a simple storage element, which we once again model as a D flip-flop. Figure 5-1 shows a schematic diagram of a generic shift register. Your initial inspection of Figure 5-1 should reveal that there is not that much to the circuit. Upon further inspection, you should discern the following:

- The shift register is a sequential circuit that is similar to the simple register. First, we can model the n-bit shift register as a set of "n" specially connected D flip-flops. Second, all the D flip-flops in the shift register share the same clock signal, which indicates all the storage elements are acting in parallel.

- The only difference between simple register and shift registers is the notion that the individual storage elements in the flip-flops connect to each other in a special way. While simple registers had D flip-flops that had inputs and outputs connected to the outside world only, the shift register has inputs and outputs that interconnect between individual storage elements. Figure 5-1 shows that the output of one flip-flop becomes the input to the adjacent flip-flop in the shift register. This special type of connection is what makes it a shift register.

- The number of bit storage elements in a shift register generally defines shift registers. The shift register in Figure 5-1 represents a generic model of a shift register including the magic ellipsis' placed in strategic locations. Common descriptions of shift registers include "a 4-bit shift register" or "an 8-bit shift register", etc. Thus, Figure 5-1 shows an "n-bit shift register" in its most generic form.

- Often times we use the letters "SR" to refer to a shift register (not to be confused with a SR latch).

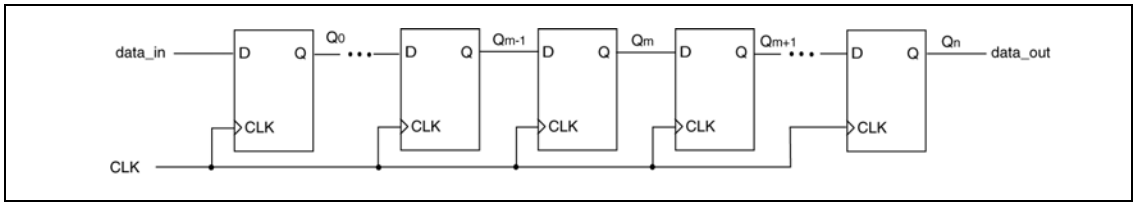


Figure 5-1: A typical n element shift register.

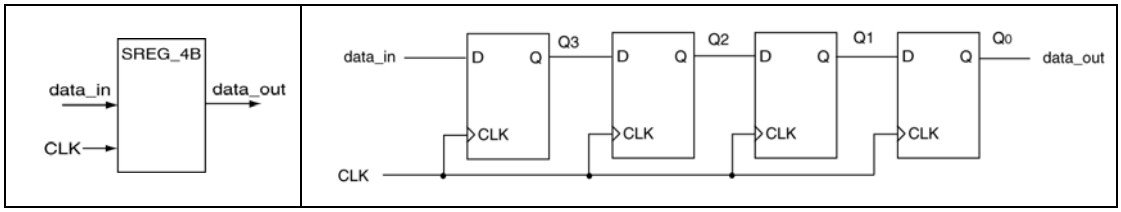
Figure 5-2 shows a comparison of block diagrams for a simple 4-bit register and a basic 4-bit shift register[3]. The important thing to notice from these diagrams is that the simple 4-bit register generally deals with "parallel" data while the basic shift register generally deals with "serial" data. What you'll find later in this chapter is that the definition of these devices starts to overlap as we add more features to the devices.

[1] Such as universal shift registers, barrel shifters, cyclic redundancy checking, bowling ball polishers…
[2] Back in the days before microcontrollers, you often saw shifters used as FSMs that acted as controllers.
[3] Keep in mind that the block diagrams show only the very basic devices for comparison purposes, which hopefully is somewhat instructive.

(a) **(b)**

Figure 5-2: A block diagram for a 4-bit simple register (a) and a basic 4-bit shift register (b).

The operation of a shift register is simple but can be somewhat tricky when you first encounter it. Figure 5-3(a) shows a schematic diagram of a 4-bit shift register while Figure 5-3 (b) shows a model of the underlying circuitry. There is not a lot to say about Figure 5-3 as the fun stuff begins when you examine a timing diagram associated with this circuit. Figure 5-4 shows an example timing diagram associated with the 4-bit shift register of Figure 5-3(b). Figure 5-4 contains some annotations to help with the following description of the shift register.



(a) **(b)**

Figure 5-3: A block diagram for a 4-bit simple register (a) and a model of the underlying circuitry of a 4-bit shift register (b).

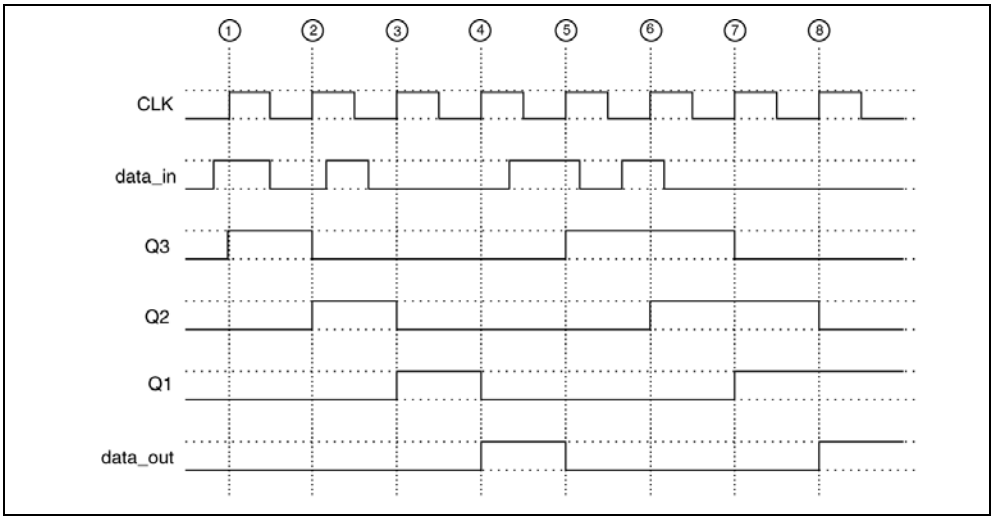


Figure 5-4: An arbitrary timing diagram associated with the shift register of Figure 5-3(b).

Here are a few items regarding Figure 5-3(b) and Figure 5-4.

- This is a 4-bit shift register, meaning that the shift register circuitry contains four storage elements. Figure 5-3(a) makes a feeble attempt to indicate this is a 4-bit shift register with the label attached to the schematic symbol.

- The schematic in Figure 5-3(b) labels each of the internal shift register signals. These labels primarily serve to help describe the operation of the basic shift register as they relate to Figure 5-4. The notation of "Qx" is typical of any circuit having flip-flops as storage elements.

- The "Qx" notation used in these figures indicates the bit positions of the storage elements in the shift register. For this example, we consider Q3 the higher order bit while Q0 (or data_out) is the lowest order bit[4]. In order to simplify this explanation, the signal "data_out" and "Q0" are the same signal.

- We consider shift registers to "shift" in either direction; that is, they shift to the left ("shift left") or shift to the right ("shift right"). The way we drew the circuit of Figure 5-3(b) makes this a right-shifting shift register; this circuit can't shift left based on its internal connections.

The notion of this circuit actually shifting something can be initially confusing. Please realize that the notion of "shifting" is primarily a term of convenience and not altogether accurate for the actual operation of the circuit (but we'll keep using it). The "thing" being shifted by the shift register of Figure 5-3(b) is the "data". Another way to look at this is that the circuit is passing in 1's and 0's from the left side of the circuit and passing them through to the right side. Figure 5-4 makes a feeble attempt at showing this shifting by way of a timing diagram. Here are some more fun things to note about the timing diagram in Figure 5-4.

- Since this is a sequential circuit, the storage elements have a state associated with them. For the timing diagram of Figure 5-4, the initial state of each storage element is '0', which is completely arbitrary.

- Since the storage elements are D flip-flops, they can only change state on the active clock edge.

- On the clock edge labeled '1', all of the flip-flops transfer the value on their inputs to their outputs. In other words, on the active clock edge, the left-most flip-flop latches "data_in"; Q3 latches into the second to the left-most flip-flop, etc. In yet another way of looking at this, on each active clock edge, each of the four D flip-flops can change state or hold their current state based on the value of the D input.

- The "data_in" input is changing at various times; the only time the input has an effect is on the active clock edge.

- Overall, if you stand back a few paces, you can see the so-called shifting action of the shift register. The individual signals are shifted versions of each other; specifically, Q3 is a shifted version of "data_in", Q2 is a shifted version of Q3, etc. Another way to look at this is that the "data_out" signal is a delayed version of the "data_in" signal. In this case, Q0 is a delayed version of Q3; the delay is three clock cycles because the pulse appearing on Q0 is the same pulse that appeared on Q3 three clock cycles earlier.

Yes, shift registers are massively powerful. Keep in mind that the right-shift operation (one shift in the right direction) is the same thing as a divide-by-two operation with truncation[5]. Also, keep in mind that the circuit

[4] Keep in mind that SRs are often used for mathematical operations; numbers generally have weights associated with the bit positions (unless you're a cave-person).

[5] Truncation means the lowest order bit is lost; a similar operation is "round-up" where the value of the lowest order bit is "taken into account" and your weeds are killed at the same time.

Figure 5-3(b) is overly simple. In most shift register circuits there are other features such as most of the ones that follow. We'll discuss these in upcoming sections.

- parallel clearing or parallel setting of storage elements
- parallel loading of values to the storage elements
- Shifting left or shift right operations
- Multiple shifts on one clock edge
- Automatic bowling score feature

A few final comments regarding basic shift registers… shift register are amazingly straightforward to model in VHDL. Even shift registers that have all the bells and whistles take about zero-time to model if you understand the basics of how VHDL models sequential circuits. Figure 5-5 shows a simple no-feature VHDL model of the shift register shown in Figure 5-3(b). As you can see from Figure 5-5, the VDHL model for this simple shift register is very similar to the VHDL model for a D flip-flop (although there is one interesting trick in this model). Keep in mind that there are many different ways to model shift registers in VHDL; the model shown in Figure 5-5 is not necessarily the best way[6]. The VHDL model of the basic shift register has a few interesting features worth noting.

- The single-bit storage elements associated with the basic shift register are not overly apparent. It turns out that the storage elements are associated with the "s_D" signal declaration, which is 4-bit signal. The VHDL model induces memory for this signal by the fact that the "if" statement in the process does not contain an associated "else". You've seen this before and you'll see it again.

- The VHDL model accesses only three-bits of the "s_D" signal by using the "downto" operator. This is typical VHDL vernacular and you'll for sure see this again.

- The "s_D" signal essentially "retains its value" across various executions of the process statement. Relative to computer science, signals are similar to static variables (variables stored memory as opposed to being stored on the stack[7]). In a later chapter, we'll discuss VHDL constructs that are similar to local variables of computer programming fame (variables stored on the stack).

[6] But I certainly don't care because I sleep well at night knowing that the VHDL synthesizer is going to take care of the details for me.
[7] Don't worry if this does not make sense; it's computer science stuff and they design that stuff to be confusing.

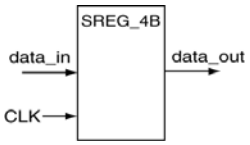
```
-----------------------------------------------------------------
-- generic 4-bit right-shifting shift register
-----------------------------------------------------------------
entity my_sr is
    port (   DATA_IN : in std_logic;
            DATA_OUT : out std_logic;
                 CLK : in std_logic);
end my_sr;

architecture my_sr of my_sr is
   signal s_D : std_logic_vector(3 downto 0);
begin

   process (CLK,DATA_IN)
   begin
      if (rising_edge(CLK)) then
         s_D <= DATA_IN & s_D(3 downto 1);
      end if;
   end process;

   DATA_OUT <= s_D(0);

end my_sr;
```

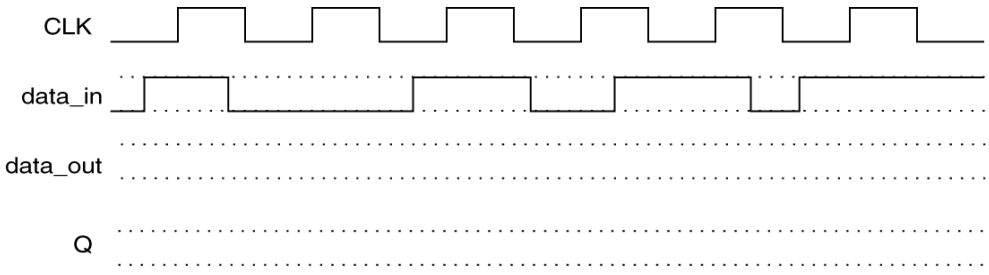
Figure 5-5: VHDL model for a basic 4-bit shift register.

Another issue that usually surrounds shift registers is the notion of cascadeabilitly. If you're actually unfortunate enough to need to use shift registers on discrete ICs, you may have to use a bunch of them to actually obtain the data width that you need. For example, if you need a 64-bit SR, and all you have to work with are ICs containing 8-bit shift registers, you'll need to cascade[8] eight 8-bit shift registers in order to create a 64-bit SR. Sad as it seems, people really used to do things this way.

Example 5-1: A Simple Shift Register Timing Diagram

Using the block diagram on the right to complete the timing diagram provided below. Consider the circuit to be a 4-bit shift register (shifts from left to right) that is active on the rising-edge triggered of the clock signal. Consider the line labeled "Q" to represent the 4-bit value stored by the shift register. Assume the "data_out" signal is the LSB of Q. Assume the initial value stored by the shift register is 0x8. Ignore all propagation delay issues.





[8] In this context "cascade" is a fancy way of saying "connect up the part properly".

Solution: The interesting thing about this example is that the problem statement provides the initial value stored in the shift register. The other interesting thing is that the problem asks for what is being stored in the shift register despite the fact that only one-bit of shift register's contents appears as an output (the "data_out" signal is the output of the LSB).

Figure 5-6 shows the solution to this example. I'm omitting lots of written description, as this is a great problem for you to work through yourself. Keep in mind that this is a right-shifting shift register, which means the "data_in" signal is the MSB of the shift register while the "data_out" signal is the LSB. Another happy thing to note about this problem is the fact that the shift register is dividing the current shift register contents by a factor of two (with truncation) when the "data_in" signal is a '0'. Convince yourself of this because this is a massively important and useful feature of shift registers.

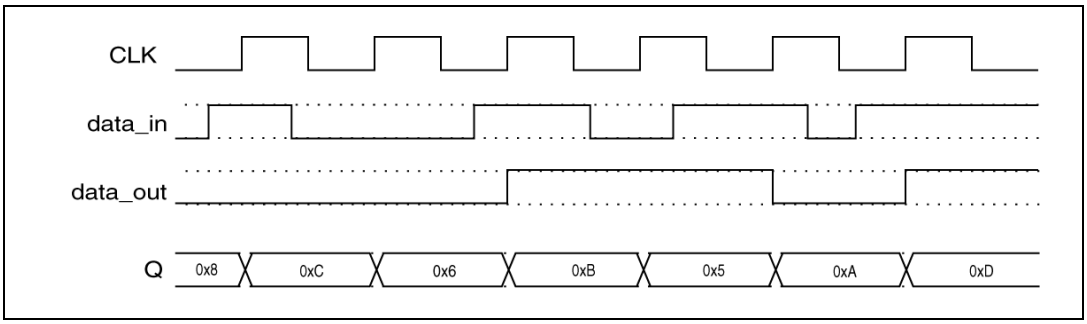


Figure 5-6: The solution to Example 5-2.

5.3.1 Universal Shift Registers

Shift registers that only shift in one direction are not overly useful in digital-land. Most shift registers do many more operations such as shift left, shift right, parallel load, parallel clear, hold (don't change state), pick up the spare, etc. The term in digital-land for shift registers containing features such as these is "universal shift register", or "USR". There is no one definition for universal shift registers; the only thing the term means is that you're dealing with some sort of shift register that does more than shift in one direction. From that point, you need to consult the datasheet or designer as to what exactly the device does.

In terms of digital design, you have the ability to easily design just about anything using VHDL. Because VHDL is so powerful, you can thus design devices at just about any level of abstraction. The following example problem picks a certain level of abstraction in its implementation of a universal shift register. There may be better ways to implement universal shift register, but the following approach is relatively efficient and somewhat instructive. Once again, we highlight the power of VHDL behavioral modeling by the fact that no matter what features your universal shift register requires, it's rather straightforward to model in VHDL. The following quick example hopefully shows that.

Example 5-2: A Simple Universal Shift Register

Provide a VHDL model for an 8-bit universal shift register that supports the following operational characteristics. For this problem, assume that all shift register operations are synchronous (meaning they happen at the same time as the rising clock edge). The shift register's output should be only an 8-bit bundle that indicates the current state of the shift register.

- Hold
- Shift right
- Shift left
- Parallel load

Solution: The first step in this problem is to understand all of the features requested by the problem. The following list describes these features, in case you were actually wondering.

- Hold: This operation means that the shift register's contents do not change state on active clock edge.
- Shift Right: This is a typical shift right operation; this typically means that there needs to be an input into the shift register from the left side.
- Shift Left: This is typical shift left operation; this typically means that there needs to be an input into the shift register from the right side[9].
- Parallel Load: This implies that there needs to be an 8-bit bundle input that simultaneously loads all the shift register elements.

From the above clarifications, we now know two types of information: the number and widths of the inputs and outputs required to complete this problem. Specifically, from this list of happy stuff, we can generate the block diagram of Figure 5-7. By the way, drawing a block diagram is still a great place to start problems such as they help you understand the problem and help you generate the VHDL entity. The following lists some other fun stuff.

- The shift register has four unique operations: hold, shift-right, shift-left, and parallel load. This means we somehow need to control which operation actually occur. We do this by adding a control signal that "selects" the desired operation. This signal is an input to the shift register and allows some external circuit to control the shift register. Since the shift register has four operations, we need a two-bit control signal to select an operation.

- We know all the inputs and outputs to the shift register. The problem states the outputs; they comprise of the values held in the shift register storage elements. The inputs include a 2-bit operation select signal, a 1-bit input for shift-left operations, a 1-bit input for shift-right operations, an 8-bit bundle for parallel loads, and a lively clock signal.

[9] You could also use the same signal for inputting signals for either shift-left or shift-right operations. The problem did not state how to do this so we have arbitrarily decided to have an input for both "sides".

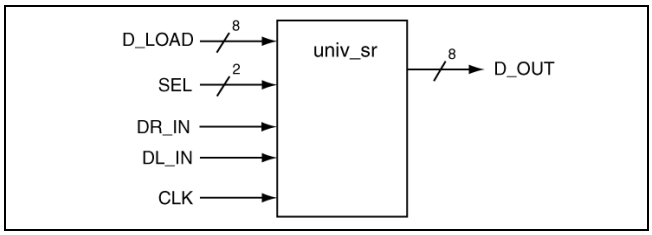


Figure 5-7: A black box diagram of the universal shift register.

Allow me to blather on about this before we get to the VHDL model. Figure 5-8 repeats Figure 5-1 for your viewing convenience; this diagram once again shows a generic schematic for a simple right-shifting shift register. The way you should think about the hardware-based solution to this problem is to imagine that each shift register storage element is now going to have to decide upon what value is going to be loaded on the next active clock edge.

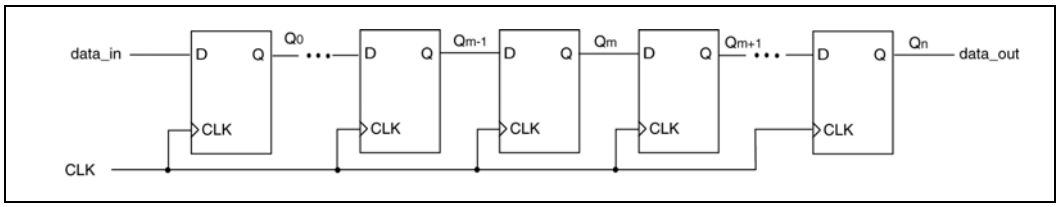


Figure 5-8: A typical n element shift register.

When you hear the word "decision" in digital design-land, you should think "MUX". If you think about it in this manner, it sure seems as if each storage element is now going to have its own MUX to decide which value is going to be loaded to the storage element. The notion here is each shift register storage element needs to decide which signal will load into the element. Figure 5-9 shows the schematic for the single shift register storage element that you're probably imagining.

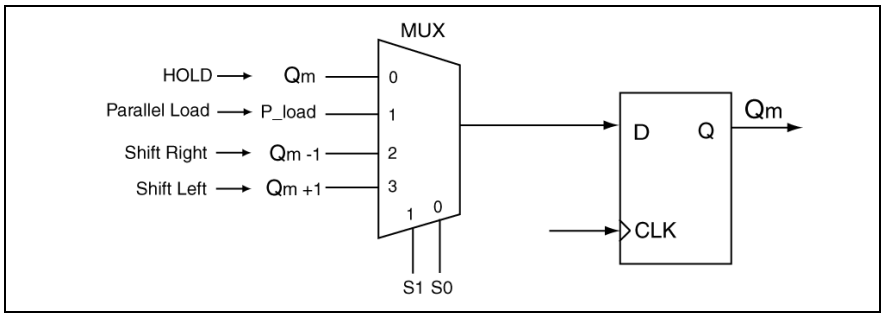


Figure 5-9: A shift register element with an attached MUX for data selection.

Figure 5-9 shows a 4:1 MUX with two control signals. The control signal selects between four different signals to load into the storage element in order to satisfy the problem. Figure 5-9 shows the MUX data signals; the choice of MUX input index was arbitrary.

- **Qm** (0): The input to the D flip-flop is the current output of the D flip-flop in question. Qm is an internal signal and ensures that the storage element does not change state by "reloading" its current value. In other words, the present state of the D flip-flop becomes the next state.

- **P_load** (1): The input to the D flip-flop is the appropriate signal from the parallel loading bundle input.

- **Qm-1** (2): The input is part of a shift right operation, which indicates the input to this storage element is the first storage element to the left of this storage element (this is confusing; check out Figure 5-8 for the details for the subscripted numbers).

- **Qm+1** (3): The input is part of a shift left operation, which indicates the input to a storage element is the first storage element to the right of this storage element (check out Figure 5-8 for clarification).

Table 5.1 summarizes the information in the previous list. So, all we need to do from here is to put a bunch of these in a row and call it a universal shift register? We could proceed with at this level, but let's instead bump up a level of abstraction in order to complete this problem. We need to use VHDL, so we may as well use it to make solving this problem as easy as possible.

S1	S0	D	Comment
0	0	Q_m	hold
0	1	P_load	parallel load
1	0	Q_{m-1}	shift right
1	1	Q_{m+1}	shift left

Table 5.1: Summary of the SR element functionality.

The modeling of a universal shift register using VHDL is straightforward and instructive. It's only slightly more complicated than just a simple D flip-flop (or simple register). To put it another way, if you understand how to generate a D flip-flop using VHDL, you'll understand the VHDL implementation of a universal shift register. Figure 5-10 once again shows the block diagram we'll use while Figure 5-11 shows the associated VHDL model.

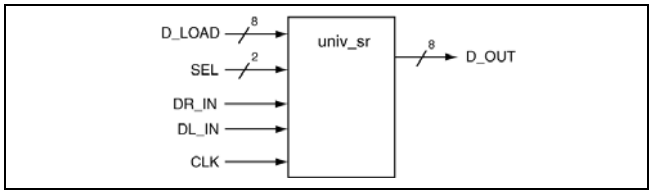


Figure 5-10: A black box diagram of the universal shift register.

```
-----------------------------------------------------------------
-- Model for a universal shift register
-----------------------------------------------------------------
entity univ_sr is
    port (    SEL : in std_logic_vector(1 downto 0);
            P_LOAD : in std_logic_vector(7 downto 0);
             D_OUT : out std_logic_vector(7 downto 0);
               CLK : in std_logic;
            DR_IN : in std_logic;  -- input for shift left
            DL_IN : in std_logic); -- input for shift right
end univ_sr;

architecture my_sr of univ_sr is
   signal s_D : std_logic_vector(7 downto 0);
begin

   process (CLK,SEL,DR_IN,DL_IN,P_LOAD)
   begin
      if (rising_edge(CLK)) then

         case SEL is
            -- do nothing (don't change state) --------------
            when "00" =>  s_D <= s_D;

            -- parallel load --------------------------------
            when "01" =>  s_D <= P_LOAD;

            -- shift right ----------------------------------
            when "10" =>  s_D <= DL_IN & s_D(7 downto 1);

            -- shift left -----------------------------------
            when "11" =>  s_D <= s_D(6 downto 0) & DR_IN;

            -- default case ---------------------------------
            when others => s_D <= "00000000";
         end case;

      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

Figure 5-11: VHDL model for the universal shift register.

The VHDL model in Figure 5-11 shows some cool stuff, as the list below happily mentions:

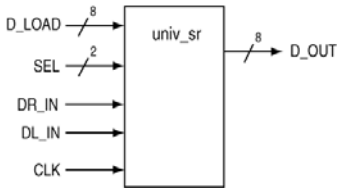
- The approach this model takes is to “synthesize” the correct eight bits based on the value of the SEL input; the rising edge of the clock synchronizes changes to the shift register storage elements.

- There are a couple of instances of the “&” operator, which is the concatenation operation in VHDL. The operator concatenates two signals (or parts of two signals) together in order to synthesize the correct output signal based on the selected operation.

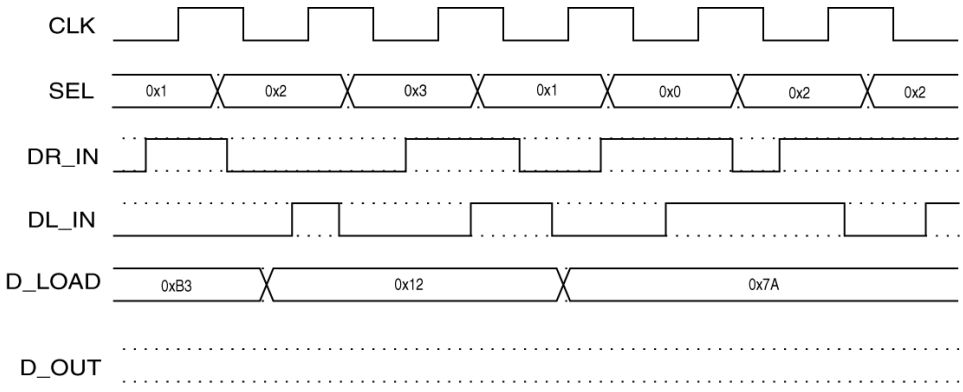
- The model takes the approach of assigning a temporary signal inside the process; once the process terminates, an assignment to the “s_D” signal causes the subsequent assignment of this signal to “D_OUT”, which is the parallel output of the shift register. This is not the only approach you can take, but it is the definitely the most clear.

Example 5-3: Universal Shift Regsiter Timing Diagram

The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:



- SEL = “00”: hold
- SEL = “01”: parallel load of D_LOAD data
- SEL = “10”: right shift; DL_IN input on left
- SEL = “11”: left shift: DR_IN input on right
- All operations are synchronized to the rising edge of the CLK signal.
- Propagation delays are negligent.
- Initial D_OUT value is 0x45



Solution: The first step in any problem involving a sequential circuit is to establish the initial state of the storage elements. This problem states that the initial value of D_OUT value is 0x45; this value is the initial state of the shift register.

From there, a good approach to problems such as these is to list what actions the SEL signal is selecting throughout the timing diagrams. Figure 5-12 shows a partially annotated timing diagram highlighting the operations selected by the SEL signal. Note that we synchronize all annotations with the rising clock edge.

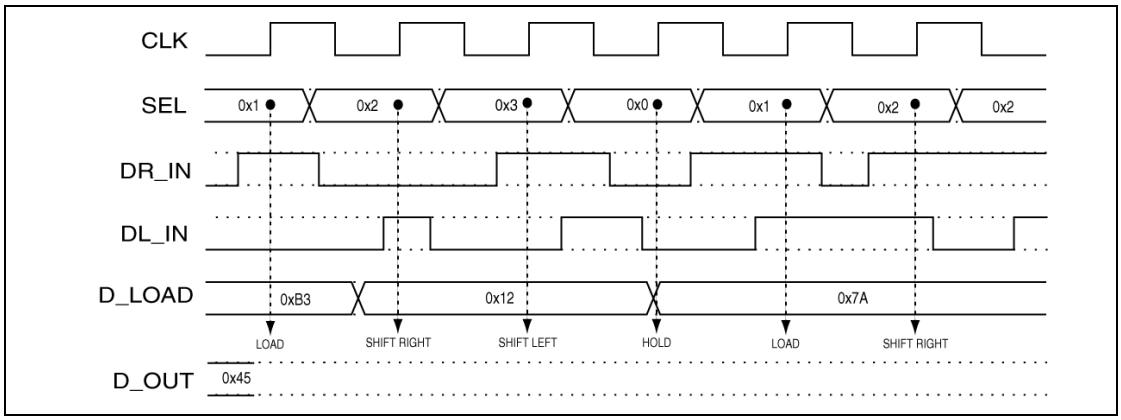


Figure 5-12: A black box diagram of the universal shift register.

Figure 5-13 shows the final timing diagram. As you can see, most of the changes in the DR_IN, DL_IN, and D_LOAD signals have no effect on the final output. The important thing to do for this problem is to verify for yourself that each of the values in the D_OUT is correct.

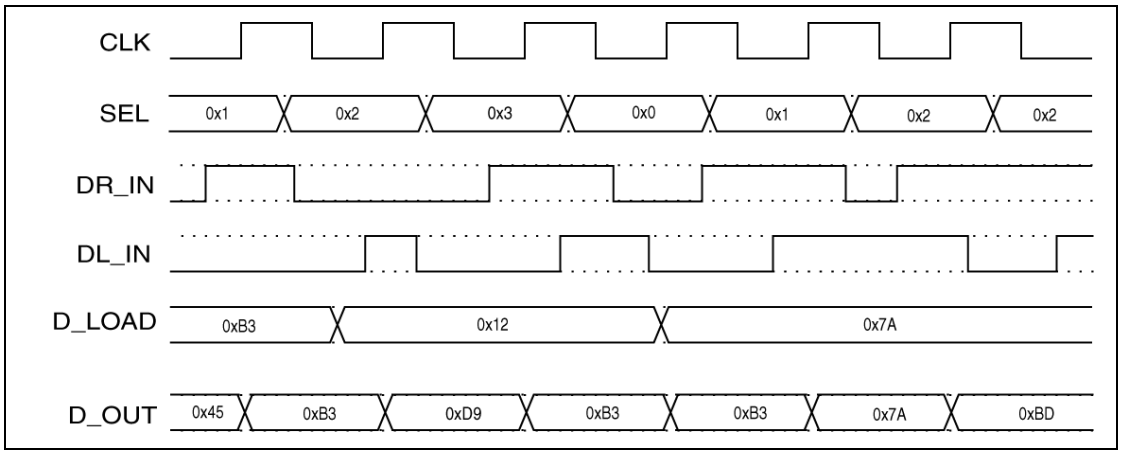


Figure 5-13: A black box diagram of the universal shift register.

5.3.2 Barrel Shifters

Once you get past the notion that shift registers do some actions that appear to be similar to "shifts", you'll find that other circuits out there do similar shifting operations. One of the common operations out there is a "barrel shift". The operation of barrel shifters is straightforward as it's simply an extension of simple shifting operations. While simple shift registers only performed one shift per clock cycle, barrel shifters are effectively capable of performing more than one shift per clock cycle. As you would imagine, barrel shifters can shift either left or right.

The key to understanding barrel shifters is realizing the main reason they exist. Keep in mind that shift registers contain "bits" which generally represent binary numbers. The notion of shifting left and right are associated with multiplying by two (left shift) or dividing (right shift) by two. Thus, barrel shifters are then associated with multiplying and dividing by "powers of two" (such as 4, 8, 16, 32, etc.). What these operations provide are super-fast (namely, one clock cycle) multiply and divide operations. As you continue in digital stuff and/or computer programming, you'll find that multiplying and dividing binary numbers is relatively

time consuming relative to other computer operations (such as logic operations). Barrel shifters provide a cheap and fast, although somewhat limited alternative.

We commonly use barrel shifters in arithmetic applications where we do not require 100% accuracy of results. For example, there is always a big push to have your circuit perform "integer-based math" because working with integers is much less "computationally expensive" than working with other options such as "floating point numbers". A good example of this is with non-professional cameras such as the ones on your cell phones. Because we partially judge cameras on these devices by their operational speed (such as how fast you can take pictures[10]), they generally use integer math. Using integer math causes you to lose some precision, but your eyes will never know the difference. All you know is that your tiny hand-held device is able to take high definition movies and do so without significant delay. Big wup.

Table 5.2 shows two examples barrel shifting operations. Both of these examples use an 8-bit value; the top example is the value before the active clock edge while the bottom value is the value after the active clock edge. The examples show both a starting and ending point for the barrel shifting operation described by the particular row in the table. The (a) row shows a 2x right barrel shift that arbitrarily inputs 0's on the left side of the register. The (b) row shows a 2x left barrel shift that arbitrarily inputs 1's from the right side of the register. The operation in the (a) row represents a divide by two; the operation in the bottom row is one the many open mysteries in this world.

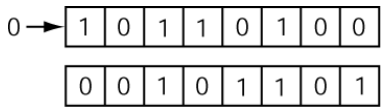
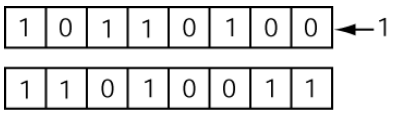
	Description	Example
(a)	barrel shift right 2x; stuff in a two 0's from the left side.	
(b)	barrel shift left 2x; stuff in a two 1's from the right side.	

Table 5.2: Examples of possible barrel shifting operations.

The examples in Table 5.2 are arbitrarily barrel shift of "2x". This syntax refers to the notion that the barrel shifter is "shifting two times" in one clock cycle. The truth is that it is only shifting one time, which implies there are connections each shift register element and the element that is two shift register elements away from the current element. As you can probably imagine, the barrel shifter requires the proper signal routing in order to accomplish this shift. As a result, barrel shifters out in digital-land are typically limited by the different flavors of barrel shifts (such as "2x") and shift directions that they can perform. Barrel shifters in these applications are typically associated with specific mathematical operations and truly don't have the general need to perform every possible shift length. Recall that for every barrel shift requires extra routing resources, which are generally not cheap in digital-land.

5.3.3 Other Shift Register-Type Features

But wait… it gets better: there are even more common shifting operations out there in digital-land. Two more of the common shifting operations are rotates and arithmetic shifts. These operations are also simple in their basic states[11]. Rotate operations can be useful in many applications, though there is not one slam-dunk great

[10] In reality, there is a significant amount of processing taking place for even the most basic digital photograph.

[11] The truth is that it can get really ugly out there. You many need to combine operations with as "barrel rotates" or "barrel arithmetic shift", or some type of shift to enhance your bowling skills. We won't go there in this chapter.

example I can think of; in theory, these operations fall into the category of "bit tweaking". Arithmetic shift operations are similar to simple shift operations but can work better with signed binary numbers.

Rotate operations include rotate left or a rotate right with the actual shifting occurring on the active clock edge. The notion with rotate-type shifts is that no bits from the original register values are lost by "shifting them out" of the register as was the case with simple shift registers. Specially, for a rotate right operation, the LSB of the register becomes the new MSB while all other bits are shifted one position to the right. For a rotate left operation, the MSB of the register becomes the new LSB while all other bits in the register are shifted one position to the left.

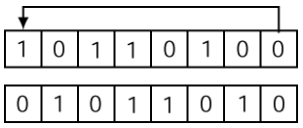
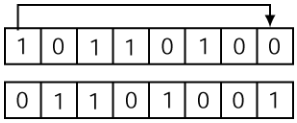
	Description	Example
(a)	rotate right; the LSB is transferred to the MSB;	
(b)	rotate left; the MSB transfers to the LSB.	

Table 5.3: Examples of rotate-type shifts.

Arithmetic shifts are similar to simple shifts in their ability to perform mathematical operations[12]. The key different is that arithmetic shifts work with signed binary number and preserved the "signedness" of the value they operate on. For an arithmetic shift left operation, the value of the sign bit does not change because of the shift. Thus, the left shift operation retains the sign of the number as well as the ability to perform fast multiplication with the left shift operation. For an arithmetic shift right operation, we both retain the sign bit as a sign bit and propagate the sign bit to the right with each shift. This sounds somewhat strange, but it truly both retains the sign of the value in the register as well as performing a fast division operation. I suggest working through a few examples on your own.

[12] When you read this paragraph, recall that we represent signed binary numbers using 2's complement notation, AKA, "diminished radix complement" notation.

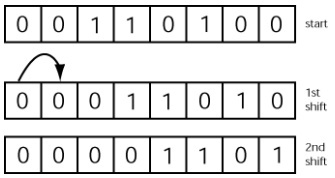
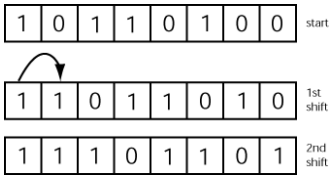
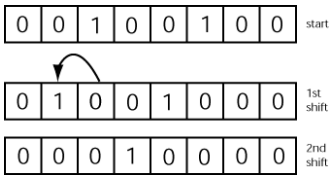
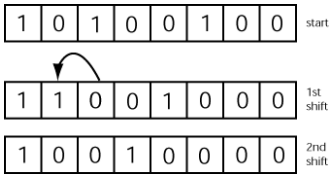
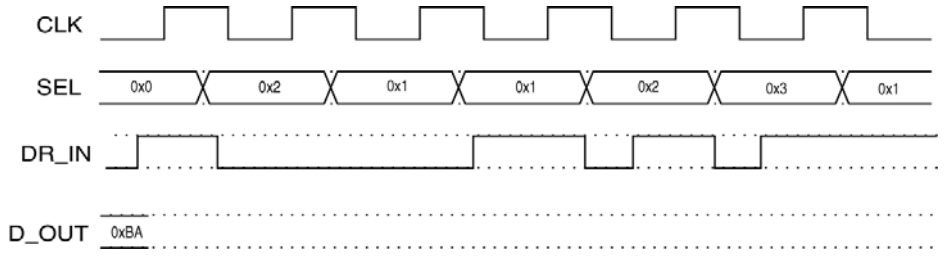
	Description	Example
(a)	An arithmetic shift right of a positive number in 2's complement form; the operation copies the sign bit from sign-bit position to the next bit on the right with each shift. This is a divide by two on a signed number (positive).	
(b)	An arithmetic shift right of a negative number in 2's complement form; the sign bit is copied from sign-bit position to the next bit on the right with each shift (the sign bit remains unchanged). This is a divide by two on a signed number (negative).	
(c)	An arithmetic shift left on a positive value in 2's complement form. The left shift does not alter the sign; all other bits shift left and the operation arbitrarily stuffs a '0' into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a signed number (positive).	
(d)	An arithmetic shift left on a negative value in 2's complement form. The left shift does not alter the sign bit; all other bits shift left and the operation arbitrarily stuffs a '0' into the LSB position. The bit adjacent to the sign bit shifts left into nowhere land. This is a multiply by two on a signed number (positive).	

Table 5.4: Examples of many flavors of arithmetic shifts.

Example 5-4: A Shifting and Rotating Circuit

Using the following timing diagram, provide a VHDL model of an 8-bit shift register that performs the operations listed below. Make sure you also complete the timing diagram and provide a block diagram of the final circuit. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need to make in order to get past yet another poorly worded example problem. Assume the DR_IN signal is the bit that is an input on the right for shift left operations while shift right operations utilize the sign bit for an input. Assume D_OUT represents the 8-bit value stored by the shift register.

- SEL = "00": arithmetic shift right
- SEL = "01": arithmetic shift left
- SEL = "10": rotate right
- SEL = "11": rotate left



Solution: The first step in any problem that does not provide a black box diagram is to generate the black box diagram. From the problem statement we can see that the circuit's input are a clock signal (CLK), a selection signal (SEL), and a bit input signal (DR_IN). The only output of the circuit is the D_OUT signal, which represents the contents of the shift register. Figure 5-14 shows the final block diagram for this example problem.

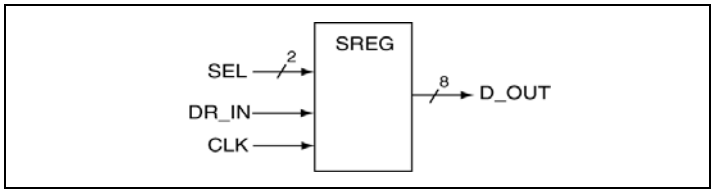


Figure 5-14: A black box diagram of the universal shift register of Example 5-4.

The next step is to annotate the provided timing diagram to explicitly show (in English) the operations selected by the SEL signal. This step is not necessary, but it ensures the mistakes you make are of the intelligent type rather than dumbtarted type. Figure 5-15 shows this intermediate helper step.

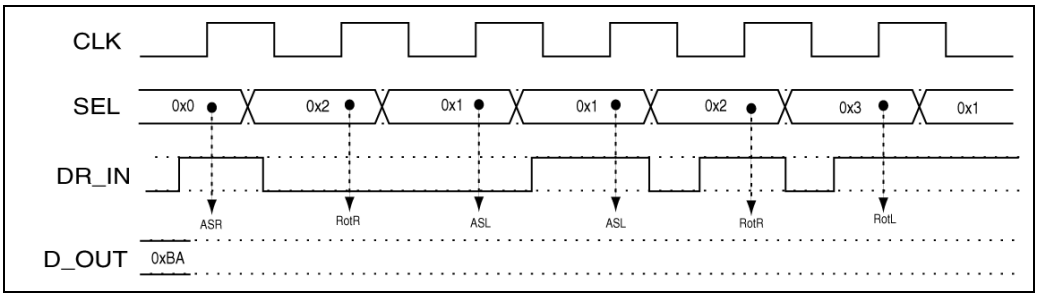


Figure 5-15: A black box diagram of the universal shift register of Example 5-4.

Without too much verbage, Figure 5-16 shows the final timing diagram solution to Example 5-4. One thing to note about this problem is that the circuit only uses the DR_IN input for arithmetic shift left operations.

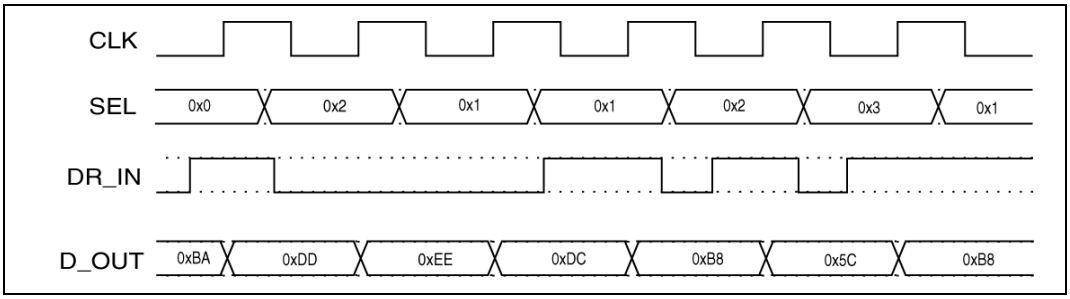


Figure 5-16: A black box diagram of the universal shift register of Example 5-4.

This example problem also states that we need a VHDL model also. This solution uses Figure 5-14 and the problem description as an aid in generating the VHDL model; Figure 5-17 shows the final VHDL model. The only worthy comment to make about this model is that the VHDL code models the "next values" of the shift register using a combination of the previous state of the shift register, the sign bit, and the DR_IN input. Liberal use of the concatenation operator also helps this model.

```
-----------------------------------------------------------------
-- Yet another shift register model
-----------------------------------------------------------------
entity sreg is
    port (    SEL : in std_logic_vector(1 downto 0);
            D_OUT : out std_logic_vector(7 downto 0);
              CLK : in std_logic;
            DR_IN : in std_logic); -- input for shift left
end sreg;

architecture my_sr of sreg is
   signal s_D : std_logic_vector(7 downto 0);
begin
   process (CLK,SEL,DR_IN)
   begin
      if (rising_edge(CLK)) then
         case SEL is
            -- arithmetic shift right
            when "00" =>  s_D <= s_D(7) & s_D(7 downto 1);

            -- arithmetic shift left
            when "01" =>  s_D <= s_D(7) & s_D(5 downto 0) & DR_IN;

            -- rotate right
            when "10" =>  s_D <= s_D(0) & s_D(7 downto 1);

            -- rotate left
            when "11" =>  s_D <= s_D(6 downto 0) & s_D(7);

            when others => s_D <= (others -> '0');
         end case;
      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

Figure 5-17: The final VHDL model for Example 5-4.

5.4 Counters: Yet Another Register Flavor

Registers… it's hard to underscore their popularity in digital design. Another massively common register is the counter. In its simplest form, a counter is a register that "counts", but also retains all the characteristics of a register (such as operations synchronized with a clock signal). While this sounds straightforward, the reality is that people make many assumptions when they use the term "counter". The list below describes these assumptions.

- Unless otherwise stated, a counter is actually a binary counter, meaning that it counts in binary. This is an important distinction because there is also the notion of a decade counter (we'll talk about that later), which does not count using a complete binary sequence. You can design a counter to count in any sequence, but we assume a normal binary sequence unless stated otherwise[13].

- Related to the last issue is the notion that a counter only counts "up" unless otherwise stated. There are also counters that count down also (more on this later).

- Once again related to the last issues is the notion that counters count up by a value of '1' on each clock cycle, unless stated otherwise. This means that a '1' is added (arithmetic addition) to the current count value on each clock edge. We commonly refer to the notion of counting up by '1' as it relates

[13] Keep in mind that we previously designed counters using FSMs; many of these counters had wacky and pointless counting sequences.

to a counter as an increment. You can easily design counters that count up (or down) by any value. We use the word "decrement" for counters that subtract one from the current value on every clock edge.

When counters are the topic of discussion, you may hear many new and unusual words. The idea of counters is straightforward, meaning, I can't think of any new and amazing things to say about them that we have not already said. In addition, we've worked with counter in the context of FSMs many chapters ago. The approach I'll take here is to define and describe every word and/or term I've ever heard used in the context of counters and then do a few example problems. In truth, counters used to be a big deal back when you had to design them yourself using discrete logic. Now, discrete ICs have many flavors of counters, and more importantly, VHDL makes the modeling of counters almost trivial. I'll save all the verbage and remain at a high level of abstraction in regards to counter design.

When you say the word counter, it has a few standard connotations that you can assume are true unless told otherwise. The following list describes even more assumptions made when dealing with counters.

- Counters always refer to a sequential circuit. There are combinatorial counters out there, but they are somewhat rare and painful to think about (but interesting all the same).
- An active clock edge synchronizes a counter's traversing of the count sequence. Thus, there is one count value, or code-word from the count sequence at each clock cycle.
- A counter's output represents a specific and repeatable sequence of a given number of bits. This means that the sequence the counter "counts" in will not change; the bit-width of the counter won't magically change either.
- When a counter completes a traversal through its count sequence (either in the up or down direction), the counter automatically starts counting over. Many times, we refer to this notion as the fact that the counter is "circular".

Wow, those facts and definitions were so fun that we'll follow them up with a listing of vernacular and definitions typically associated with counters (and similar devices):

- Binary Counter: A counter that counts in a binary sequence. This means a 4-bit binary count sequence goes from 0-15, or 0x0 to 0xF (up direction).
- Decade Counter: A counter that counts in a binary coded decimal (BCD) sequence. This means a 4-bit decade counter will count from 0-9 (up direction).
- n-bit Counter: A counter that uses n-bits to represent each of the values in its count sequence.
- Up Counter: A counter that counts up (increments by '1', increasing count values in count sequence).
- Down Counter: A counter that counts down (decrements by '1', decreasing count values in count sequence).
- Up/Down Counter: A counter that can count either up or down according to a selection input on the device.
- Increment: An operation associated with counters where '1' is added to the current value of counter.
- Decrement: An operation associated with counters where '1' is subtracted from the current value of counter.

- Counter Overflow: The notion of a counter being incremented beyond its ability to represent values; unless otherwise stated, overflow is generally characterized as the counter transitioning from its largest representable value to its smallest value.

- Counter Underflow: The notion of a counter being decremented below its ability to represent values; unless otherwise stated, underflow is generally characterized as the counter transitioning from its smallest representable value to its largest value.

- Cascadeable: A characteristic of many digital devices such as counters and shift registers that allow you to effectively increase the overall bit-width of devices providing inputs and outputs such that you can easily interface the devices. One such output is the "ripple carry out".

- Count Enable: A signal on counters that enables the counting operation of the counter when asserted and disables the counting when not asserted.

- Ripple Carry Out (RCO): A signal typically found on counters that indicate when the counter has reached its maximum count value (for an up counter) or minimum count (for a down counter). This signal often aids in cascading multiple counter devices. Counters often use the term RCO to indicate when the counter has reached its minimum count value (for down counters).

- Parallel Load: A characteristic of a counter or shift register indicating that all the storage elements in the device can simultaneously latch external values.

- Circular: When counters overflow their maximum or minimum counts, we consider them to "overflow". Counters are typically circular meaning that when the counter reaches the maximum value, it automatically continues counting in the same direction starting at the minimum value[14].

5.4.1 A Modern Approach to Counter Design

There are many ways to design counters; the most efficient way is to model their behavior using VHDL. Let's skip over some of the older methods as they are primarily academic exercises and are not overly useful in modern digital design. Many old digital design textbooks list these older methods; if you need to know of them, get a copy of these older texts.

When I need a counter, I go right to the generic VHDL counter model I keep around for that purpose. Figure 5-18 shows my starting point when my digital designs require counters. This model contains the basic structure of a counter in addition to many of the features associated with counters. I typically start with this model because it has everything, and then remove the parts that I don't need.

We refer to the model in Figure 5-18 as an Up/Down Counter. This counter is nothing more than a counter that has a control signal that enables the counter to count either up or down. I've included a special section for this counter simply because the term is so popular. The generic counter model in Figure 5-18 contains the code that makes it into an official up/down counter. Let's do an example problem for this counter and then move on. As you'll see from examining Figure 5-18, there are some worthy things to note:

- The counter looks a lot like all the other register models we've been discussing.

- The counter overflows when it increments at its maximum value (0xFF → 0x00) and underflows when it decrements at its minimum value (0x00 → 0xFF). These operations occur automatically so there is no need to design them into the counter.

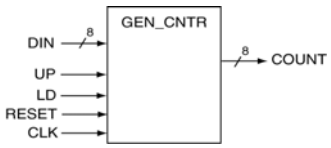
[14] This characteristic is for an up counter; the same idea is true for a down counter.

- The counter has an asynchronous reset signal. This could easily be changed to a synchronous reset if my design so desired. The current reset signal is active low.

- The parallel load signal is synchronous and takes precedence over the other basic counter operations in the model.

- The counter has a signal dedicated to the counter direction. The model indicates that if UP is asserted (a positive logic signal), the counter counts up. If the UP signal is not asserted, the counter counts down.

- The counter uses the "+" operator for incrementing and the "-" operator for subtraction. This is an advanced concept in VHDL modeling that you may or may not have encountered by now. VHDL can model mathematical operations if you include the proper library.

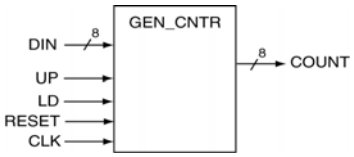


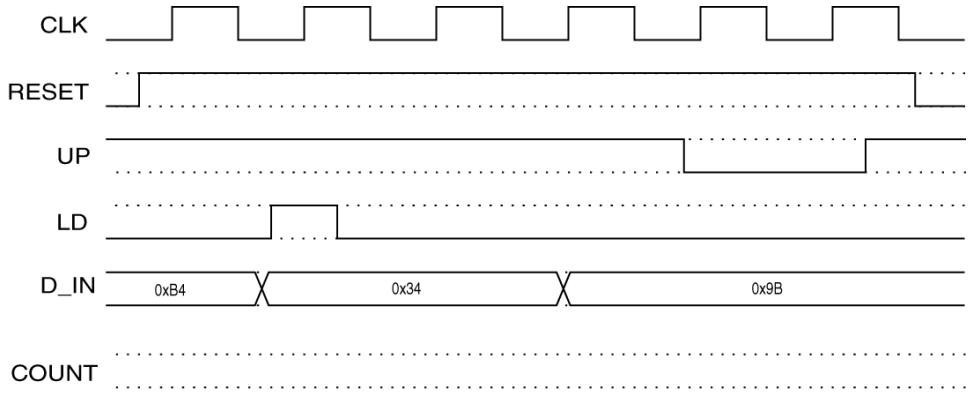

```
-------------------------------------------------------------------
-- Counter: synchronous up/down counter with asynchronous
-- reset and synchronous parallel load.
-------------------------------------------------------------------
entity GEN_CNTR is
   port ( RESET,CLK,LD,UP : in std_logic;
                     DIN : in std_logic_vector (7 downto 0);
                   COUNT : out std_logic_vector (7 downto 0));
end GEN_CNTR;

architecture my_count of GEN_CNTR is
   signal  s_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, RESET, LD, UP)
   begin
      if (RESET = '0') then
         s_cnt <= (others => '0');  -- asynchronous clear
      elsif (rising_edge(CLK)) then
         if (LD = '1') then
            s_cnt <= DIN;  -- load
         else
            if (UP = '1') then
               s_cnt <= s_cnt + 1; -- increment
            else
               s_cnt <= s_cnt - 1; -- decrement
            end if;
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

Figure 5-18: Generic VHDL model of a counter that does just about everything.

Example 5-5: Up/Down Counter Timing Diagram

The block diagram on the right shows a model of an 8-bit counter. The VHDL model associated with this block diagram appears in Figure 5-18. Use the block diagram and VHDL model to complete the following timing diagram. Assume propagation delays are negligent.





Solution: This problem attempts to show you everything interesting and useful with counters. Figure 5-20 shows the final solution to this example; the following verbage describes some of the more interesting things about the solution. In this case, the interesting things are when the output changes and what causes those changes.

1) The circuit was initially in a reset condition. On this active clock edge, the counter output increments due to the assertation of the UP signal.

2) The UP signal is still asserted, but due to the way we modeled the LD signal, it takes precedence over the LD signal. Thus, the output loads the value on the D_IN input into the counter.

3) This is an increment operation due to the assertation of the UP signal.

4) This is another increment operation due to the assertation of the UP signal.

5) This is a decrement operation since the UP signal is no longer asserted.

6) This is another decrement operation since the UP signal remains unasserted.

7) This is a register clear operation due to the assertion of the RESET signal.

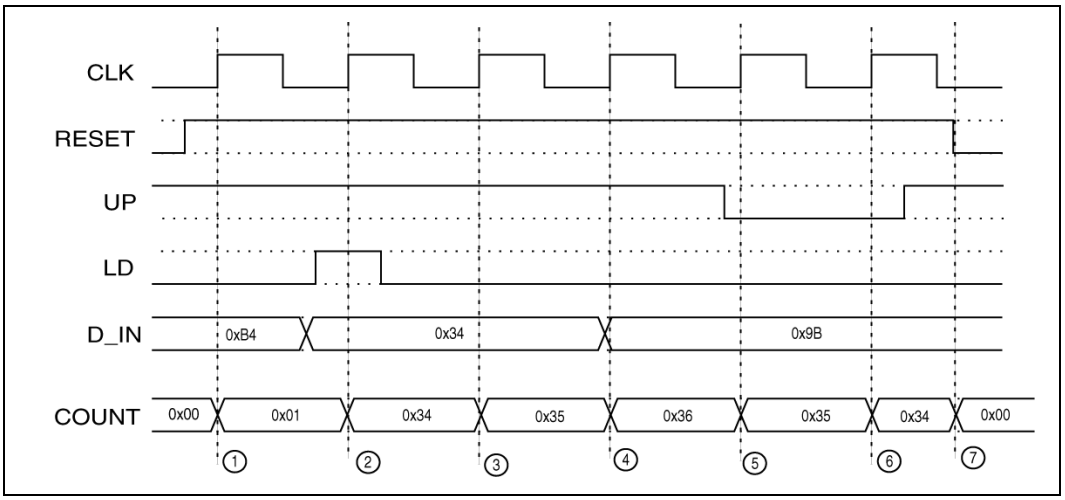


Figure 5-19: The solution (with annotiations) to Example 5-5.

Example 5-6: A Simple Up/Down Counter VHDL Model

Provide a block diagram and a VHDL model for a synchronous 8-bit up/down counter. The output of the counter should the 8-bit count only.

Solution: The first step in this problem is to find out what the problem is looking for and then generate a black box diagram. This problem wants an up/down counter, but it does not state that it needs parallel loading capabilities or any type of asynchronous presets or clears. Figure 5-20 shows the block diagram we'll work from.

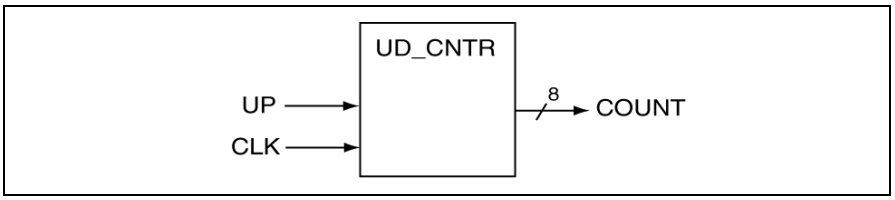


Figure 5-20: A black box diagram of the universal shift register of Example 5-4.

After generating the block diagram, we need to grab our generic counter model and modify it in order to satisfy the problem description. Figure 5-21 shows the final solution to this example problem.

```
-----------------------------------------------------------------
-- No Frills Synchronous 8-bit Up/Down Counter
-----------------------------------------------------------------
entity UD_CNTR is
   port ( CLK,UP : in std_logic;
          COUNT : out std_logic_vector (7 downto 0));
end UD_CNTR;

architecture my_count of UD_CNTR is
   signal  s_cnt : std_logic_vector(7 downto 0);
begin
   process (CLK, UP)
   begin
      if (rising_edge(CLK)) then
         if (UP = '1') then
            s_cnt <= s_cnt + 1; -- increment
         else
            s_cnt <= s_cnt - 1; -- decrement
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

Figure 5-21: VHDL model for a simple 8-bit up/down counter.

5.4.2 Decade Counters?

Another common counter you occasionally hear about is the decade counter. While we generally consider counters to be binary counters (unless specified otherwise), non-binary counters count in some sequence other than binary. A decade counter counts in a decimal sequence much like a binary coded decimal number. That is, the output of the counter shows a 4-bit binary zero through nine ("0000" → "1001") rather than a binary zero through fifteen ("0000" → "1111"). The counters can be quite useful as the non-computer portion of the world is still decimal[15]. It sounds like we need to do an example problem.

Example 5-7: Decade Counter VHDL Model

Provide a block diagram and a VHDL model for a synchronous 8-bit decade counter. The output of the counter should the 8-bit count only.

Solution: The problem describes a two-digital decimal counter; we represent each of the digits in binary using binary coded decimal numbers. This problem only has a clock input, as the problem requests no other features. Figure 5-20 shows the block diagram we'll work from.

[15] Excluding the academic administrative portion of the world which is still using stone age binary.

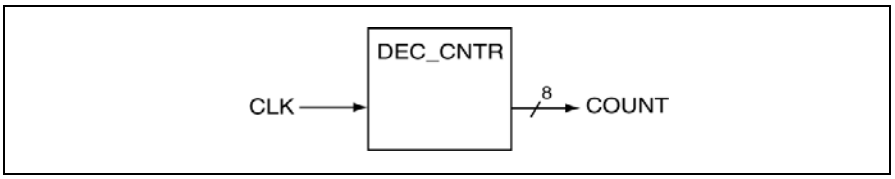


Figure 5-22: A black box diagram of the universal shift register of Example 5-4.

Figure 5-23 shows the final VHDL model for this example. This solution is by no means unique; it was the first thing I thought about when solving this problem. This problem highlights the strength of behavioral modeling in VHDL. The only problem I see with this solution is the notion that there are three levels of nesting in the "if" statements. This makes me nervous; so I will flag that in my brain and make this the first module I look at if my circuit is not working properly.

```
-----------------------------------------------------------------
-- No Frills Synchronous 2-digit Decade Counter
-----------------------------------------------------------------
entity DEC_CNTR is
   port (    CLK : in std_logic;
          COUNT : out std_logic_vector (7 downto 0));
end DEC_CNTR;

architecture my_count of DEC_CNTR is
   signal  s_cnt_tens : std_logic_vector(3 downto 0);
   signal  s_cnt_ones : std_logic_vector(3 downto 0);
begin
   s_cnt_tens <=
   process (CLK)
   begin
      if (rising_edge(CLK)) then
         if (s_cnt_ones = "1001") then
            s_cnt_ones <= "0000";
            if (s_cnt_tens = "1001") then
               s_cnt_tens <= "0000";
            else
               s_cnt_tens <= s_cnt_tens + 1; -- increment tens digit
            end if;
         else
            s_cnt_ones <= s_cnt_ones + 1;    -- increment ones digit
         end if;
      end if;
   end process;

   COUNT <= s_cnt_tens & s_cnt_ones;

end my_count;
```

Figure 5-23: VHDL model for a simple decade counter.

5.4.3 Registers: The Final Comments

The use of registers is quite common in digital. This chapter presented some of the more popular flavors of registers including counters and shift registers. The Venn diagram in Figure 5-24 shows how the various flavors of registers relate to each other.

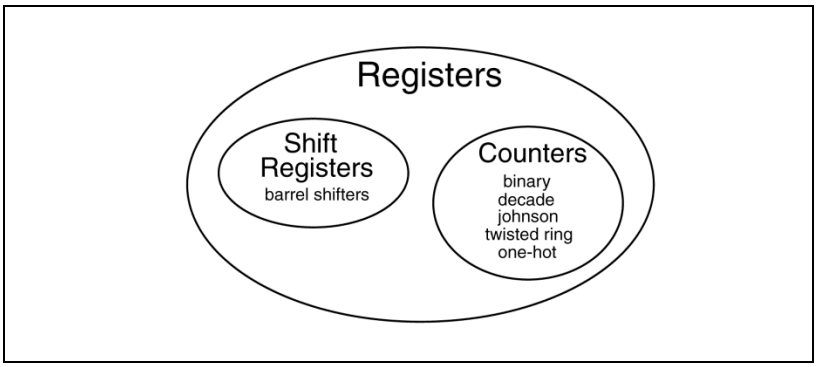
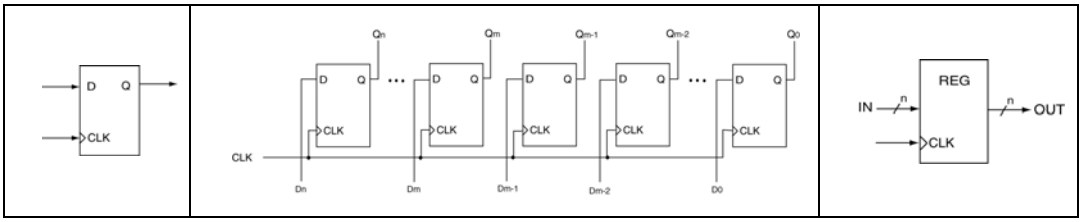


Figure 5-24: Venn diagram for the register family.

A register is nothing more than a set of bit storage elements that share a single clock signal. In other words, registers are a parallel configuration of signal bit storage elements; what makes them parallel is the fact that the individual storage element operations are generally synchronized to some event (usually a clock edge). A D flip-flop makes for an easy single bit storage element model; if you line up a bunch of D flip-flops and synchronize their actions with a clock edge, you have a register.

Once you abstract all of these matters to a higher level, you'll forever more speak about *n-bit registers*. Figure 4-17 shows the progression of this abstraction. One thing to note here is that the black box diagram of a register shown in Figure 4-17 (c) includes a clock signal. The level of abstraction here sometimes continues to the point of not including the synchronizing signal (in this case, the clock) in the block diagram. In these cases, we assume the register has a clock signal and interpret it accordingly. Additionally, you can safely assume that all registers are edge-triggered unless told otherwise.



(a) (b) (c)

Figure 5-25: The progression from D flip-flop to register block diagram for n-bit register.

The definition of the register in Figure 4-17 is general enough to encapsulate everything you know about registers up to this point. The registers we've previously looked at included several common sequential circuits such as shift registers and counters. The main difference between the many types of register is their feature set. In an attempt to show all the possibilities in one spot, Table 5.5 shows a possible breakdown of the register types and their relation to each other. Keep in mind that many of the features listed in Table 5.5 can be either synchronous or asynchronous.

Register Type	Sub-Types	Features
simple register		not much
better register		parallel load, preset, clear, load enable, cascadeability
shift register	Universal Shift Registers, Barrel Shifters	parallel load, preset, clear, load enable, shift left/right, arithmetic shift left/right, hold, rotate left/right, cascadeability
counters	Up/Down Counters, Decade Counters	parallel load, preset, clear, load enable, increment, decrement, cascadeability

Table 5.5: The feature progression of the register device.

5.5 Chapter Summary

- **Shift Registers:** Shift registers are in many ways similar to simple registers; their primary difference is with the inputs to the individual shift register storage elements. Shift registers are designed such that the data output from one shift register element becomes the data input to a contiguous element. IN this way, data is said to be "shifted through" the shift register. In general, there is one "shift" per clock cycle. Shift register operations are often used to implement fast but limited mathematical operations with single right shift being a divide-by-two and a single left shift being a multiply by two.

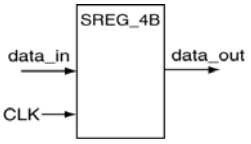
- **Universal Shift Register:** A type of shift register that performs more operations than a simple shift register. These operations can typically include both a shift left and a shift right, a parallel load, a preset and/or clear. Somewhere in here could also be arithmetic shift operations and various forms of rotate operations.

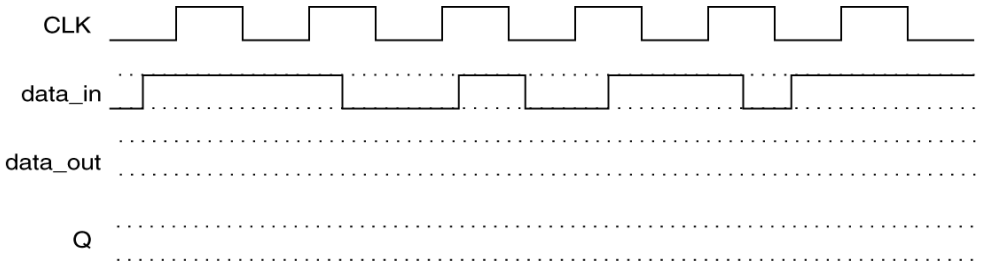
- **Barrel Shifters:** A type of shift register that performs multiple shifts on a single clock edge. In reality, barrel shifters are wired such that they can shift multiple bit locations in one clock cycle, and probably do not perform multiple shifts. Barrel shifters are useful for mathematical operations including multiplication and division by powers of two.

- **Counters:** A generic term for a device that traverses a set sequence on a given clock edge. There are many ways to design counters, the most efficient and modern approach is to use VHDL modeling. There are many types of counters out there including binary counters, up/down counters, decade counters, ring counters, Johnson counter, bowling counters, etc.

5.6 Chapter Exercises

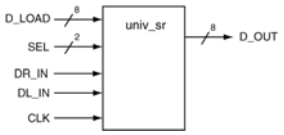
1) Use the block diagram on the right to complete the timing diagram below. Consider the circuit to be a 4-bit shift register (shifts from right-to-left) that is active on the rising-edge triggered of the clock signal. Consider the line labeled "Q" to represent the 4-bit value stored by the shift register and the "data_out" output to represent the value of the highest order bit stored by the shift register. Assume the initial value stored by the shift register is 0xC. Ignore all propagation delay issues with this circuit

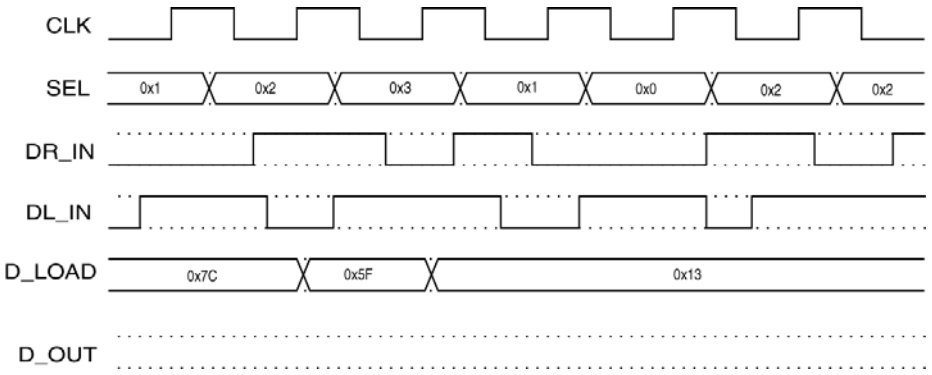




2) The block diagram on the right shows a model of a universal shift register; use this model to complete the timing diagram listed below. Consider the following:

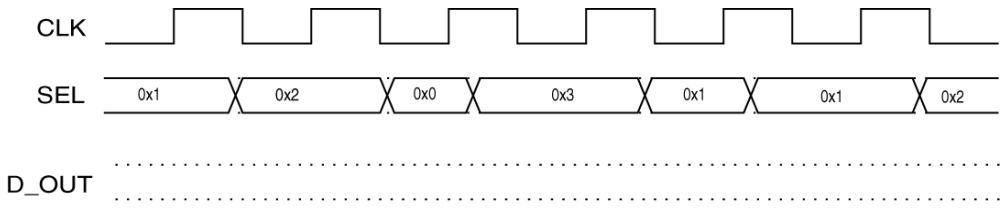
- SEL = "00": hold
- SEL = "01": parallel load of D_LOAD data
- SEL = "10": right shift; DL_IN input on left
- SEL = "11": left shift: DR_IN input on right
- The rising edge of the CLK signal synchronizes all shift register operations
- Propagation delays are negligent.
- Initial D_OUT value is 0xAB





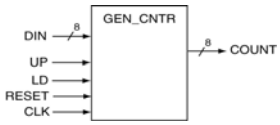
3) Using the following timing diagram, provide a VHDL model of an 8-bit shift register that performs the operations listed below. Make sure you also complete the timing diagram and provide a block diagram of the final circuit. Assume that all operations are synchronized with the rising edge of the clock signal. Assume that propagation delays are negligent. Be sure to state any other assumptions you need to make in order to complete this problem. Assume the 0x39 is the initial value stored by the shift register. Assume "D_OUT" is an 8-bit output representing the value stored by the shift register.

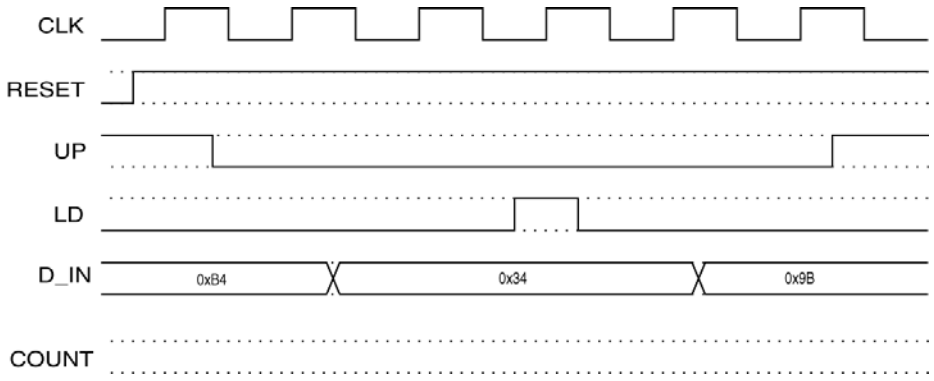
- SEL = "00": rotate right
- SEL = "01": rotate left
- SEL = "10": divide by 8 (bit stuff 0's)
- SEL = "11": multiply by 8 (bit stuff 0's)



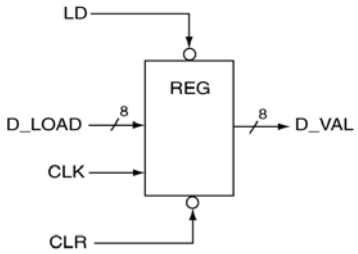
4) The block diagram on the right shows a model of an 8-bit counter. Use the following assumptions in order to complete the following timing diagram. Assume propagation delays are negligent.

- The LD input enables the DIN loading into the counter
- The RESET input is an asynchronous and active low used to reset the counter
- The COUNT output shows the current value stored by the counter
- The counter counts up when the UP input is asserted (active high) or down otherwise. All count operations are synchronous.



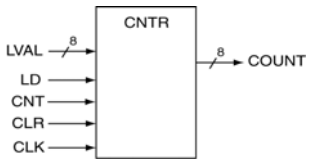


5) Provide a VHDL model that supports the black box diagram of the register on the right. Make the following assumptions for this problem.



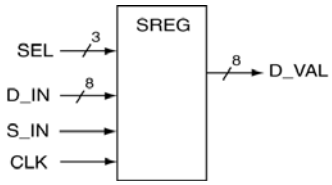
- LD synchronously loads the value of D_LOAD into the register.
- D_VAL is the 8-bit value stored in the register.
- CLR is an asynchronous input that resets the register when asserted. This input takes precedence over the LD input.

6) Provide a VHDL model that supports the black box diagram of the counter on the right. Make the following assumptions for this problem.



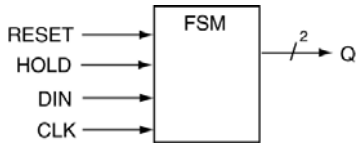
- LVAL is loaded to the counter synchronously when LD is asserted
- COUNT is the value stored in the counter.
- CLR synchronously resets the counter and takes precedence over all other inputs.
- CNT is the input of lowest precedence and instructs the counter to count up by ‘1’ if asserted and ‘2’ otherwise

7) Provide a VHDL model that supports the black box diagram of the shift register. Make the following assumptions for this problem.

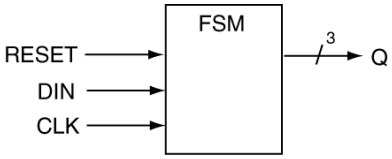


- The S_IN input is used for all single-bit shift left and shift right operations where the input value is not stated.
- D_IN is the 8-bit input value used for parallel load operations.
- The SEL input synchronously chooses the following operations
    - SEL = “000”: shift left (stuff ‘0’ on right)
    - SEL = “001”: shift left
    - SEL = “010”: shift right (stuff ‘1’ on left)
    - SEL = “011”: shift right
    - SEL = “100”: rotate left
    - SEL = “101”: divide by 4
    - SEL = “110”: divide by 16
    - SEL = “111”: multiply 8

8) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 2-bit shift register. Consider the Q output to be a 2-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs. When the HOLD input is asserted, the Q output does not change.



9) A FSM can be used to generate a shift register. For this problem, provide a state diagram that could be used to model a 3-bit shift register. Consider the Q output to be a 3-bit bus that indicates the result of the synchronous shifting action. Consider the DIN input as the bit being shifted into the shift register (shifts left to right). Consider the RESET input to be an asynchronous input that takes precedence over all other inputs.



10) The following diagram shows a circuit that is used to perform a serial-to-parallel conversion on the OP_A and OP_B input and then perform a mathematical operation. In other words, two four-bit numbers will be provided serially (LSB first) on the OP_A and OP_B inputs. The two tables below describe the MUXes and the Universal Shift Register (USR).

- Provide a state diagram that could be used to control the circuit such that it performs ***A - B*** and registers the result in REG_ACC (A & B are the parallelized versions of the OP_A & OP_B serial data). The serial to parallel conversion will initiate when the signal GO (not shown) is asserted. Minimize the number of states in your design. State any other assumptions you deem necessary.

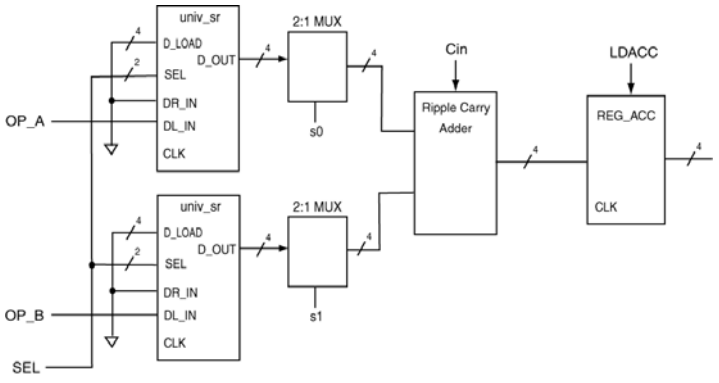
MUX description

```
if (sx = 0) then
   out <= in;
else
   out <= not
in;
end if;
```

Assumptions:

- LSB is first to arrive in serial bit stream
- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All Shift register operations are synchronous

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



11) The following diagram shows a circuit that can perform a mathematical operation. The two tables below describe the MUXes and the Universal Shift Register (USR). The registers have a synchronous load input (LD). Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.*

- If a GO signal is received (GO is not shown in diagram), the following operation is generated and the result appears on the output: **OP_OUT = (*OP_B* - *OP_A*) ÷ 16**

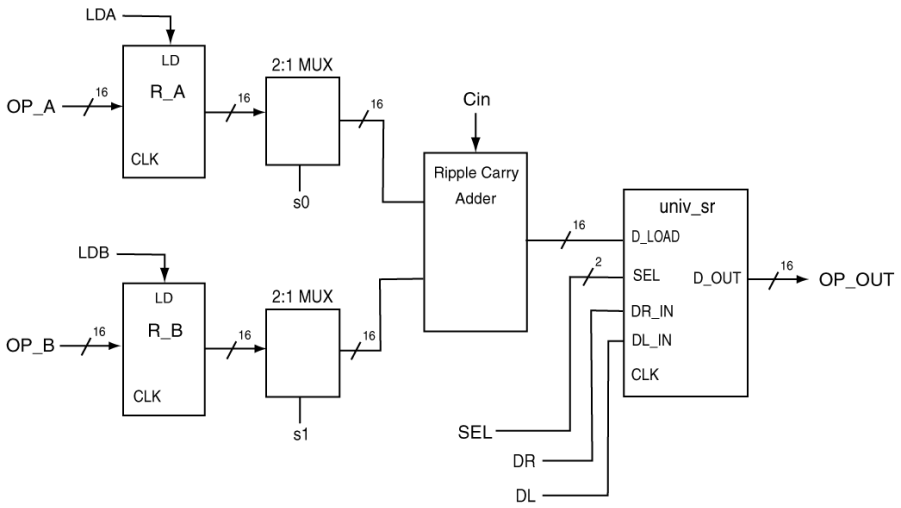
MUX description

```
if (sx = 0) then
   out <= in;
else
   out <= not
in;
end if;
```

Assumptions:

- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All Shift register operations are synchronous
- Registers (non-USR) have synchronous load inputs (LD)

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



6 Chapter: Register Transfer Notation

6.1 Introduction

Digital design is always abstracting things upwards in an effort to increase the efficiency of representing circuits. As we move on describing computer circuits, we need to come up with a new, higher-level of abstraction for representing circuits. The solution to this dilemma is what we refer to as *register transfer language (RTL)* or synonymously, *register transfer notation (RTN)*. This notation, or language, uses a simple syntax that provides a clear and concise description of a circuit. A set of register transfer language (RTL) statements can completely describe a digital system in a high-level manner, which is why it is so useful in computer design. Conversely, we can also describe a digital system by a set of RTL statements.

Main Chapter Topics

- **REGISTER TRANSFER NOTATION INTRODUCTION:** This chapter introduces the notion of register transfer notation (RTL). This chapter also discusses the motivations behind RTL and the most accepted syntax or RTL form.
- **MICROOPERATIONS:** This chapter classifies and describes basic operations that you can do with register and their relation to elementary operations.
- **DATA TRANSFER CIRCUITS:** This chapter describes three main types of data transfer circuits and provides examples of their usage, advantages, and disadvantages.

Why This Chapter is Important

This chapter is important because register transfer notation is highly useful in designing and/or describing computer operations because it provides a compact form to describe data transfers and the signals that control them.

6.2 Register Transfer Notation Specifics

Before we go here, there is one important fact that you need to keep in mind. RTL is not like VHDL in that it is not a compiled or interpreted language. With VHDL, there are many syntax-type rules you need to follow in order for your code to synthesize. The same is not true for RTL: the rules (if there really are any at all) are lax. A good analogy to this lack of rules is with the labeling of the inputs, outputs, and states of the state diagrams. The guiding principle in drawing state diagrams was to simply make it readable and understandable to anyone who was not clueless. Similar to state diagrams, since there is not absolute syntax that you can draw upon, you must be clear with the convention you use to write RTL equations. There are some RTL syntax rules that you should definitely follow in order to makes things easier to understand and share information. Equation 6. shows an example of the general form of an RTL statement.

[conditions :] destination register ← source register *[,destination register ← source register, …]*

Equation 6.1: The general form of a RTL statement.

The notation in Equation 6. notation reads as follows: ***the contents of the source register is transferred to the destination register***. Here are the important points to realize regarding this notation:

- There can be conditions associated with these transfers (as indicated by the italics in the far left of Equation 6.) which allow the transfers to occur. We aptly refer to the left-pointing arrow as the replacement operator.
- There can be multiple transfers associated with one RTL statement (once again indicated by the italics).
- A clock signal is rarely (if ever) included in RTL statements. The transfer is understood to occur on the active clock edge associated with the system, thus the system clock synchronizes all microoperations.
- The result of this transfer does not generally change the contents of the source register (and if it did, the RTL statement would list this fact).
- The register transfer operations listed in one RTL statement happen in parallel. In the context of digital circuitry, this means all the transfers happen on the same system clock edge.

Example 6.1

Draw a circuit that would implement the following RTL statement: R1 ← R2

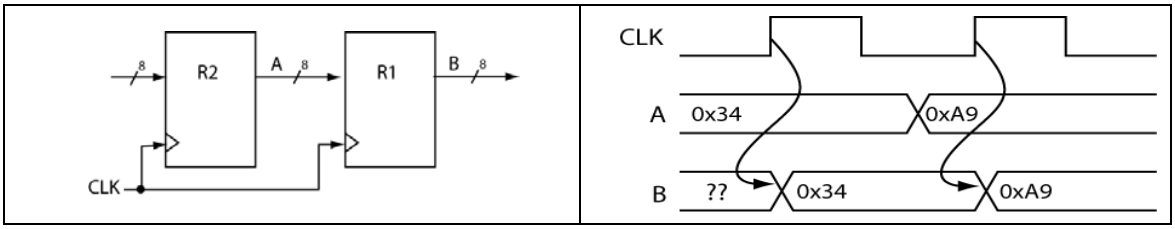
Solution: Once again, there is an interesting relationship between a RTL statement and the underlying hardware. This problem tells you what needs to be done and it is your job to design a circuit that does it. There is actually a little bit of freedom in how you do it. The RTL statement provides a guideline on what the underlying hardware should be able to do. If the hardware you generate can do it, you've got a right answer, but certainly *not the only answer*. In other words, there are generally many solutions to a given problem such as this one. There is usually a preferred solution based on the most efficient circuit so you should always strive for that option.

The thing to notice about the given RTL statement is that it lists two registers. Your final circuit will therefore have at least two registers. Also, note that there needs to be a path so data can flow from the R2 register to the R1 register. These two facts spell out the answer to the example. Churn them around in your head and you'll arrive at the circuit shown in Figure 6-1(a). The width of the data signals has been arbitrarily set to eight for this and subsequent examples.

To understand the operations of this circuit, look at the timing diagram shown in Figure 6-1(b). There is some new terminology used in this diagram that is typical of transferring data over the bus lines. The data line labeled **A** represents the output of the R2 register from Figure 6-1(a). Since this signal is a bus, we use the shorthand notation to represent all of the signals on the bus as listed in Figure 6-1(b). The "0x" notation is C programming language notation that indicates the numbers that follow it are in hexadecimal format (thus representing the eight bits of the signal). Figure 6-1(b) shows that we represent the state of the eight bits on the signals labeled **A** and **B** with this notation. Using this notation requires that the signal have the "X" notation in

the timing diagram (not the "0x") each time one or more of the signals in the bus changes. Each time an "X" appears in the timing diagram, you must list the state of all eight bits of the signal.

The timing diagram in Figure 6-1(b) shows that the signals change on each clock edge. We show this dependency by using the arrows pointing from the rising clock edge to the changing data in the **B** signal. The values on the **A** signal are arbitrary as are the times they change; what's more important is that you understand the timing and listed data transfers. Note that the value of the **A** signal changes midway between the two clock pulses but the new condition is not transferred to the **B** signal until the rising clock edge comes along.



(a) (b)

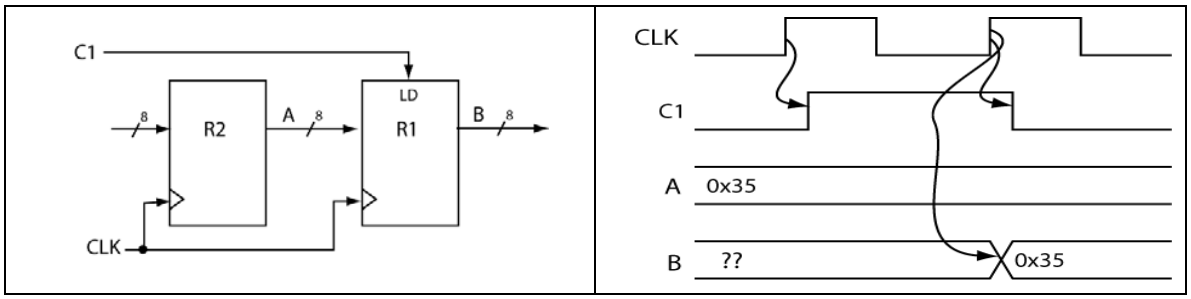
Figure 6-1: Solution and timing diagram for Example 1.

Example 6.2

Draw a circuit that would implement the following RTL statement: C1: R1 ← R2

Solution: This problem is similar to the previous problem but with a slight modification in the RTL statement. Note that in this RTL statement, there is a dependency. In other words, data is transferred from the output of R2 into the R1 register only if the **C1** signal is asserted. Most RTL statements have some type of dependency but most are more complex than this as you'll see in the final example. The circuit shown in Figure 6-2(a) provides the functionality specified by the given RTL statement. Note that the R1 register contains a **LD** input, which enables the parallel loading of data into R1 on the active clock edge.

The timing diagram shown in Figure 6-2(b) is more instructive for several reasons. First, the **C1** input is somewhat dependent upon the clock. The thought here is that the active clock edge causes a change in some other circuit that has on output that is currently driving the **C1** input. Imagine that this signal is a Moore-type output from some FSM (control unit). Note that both the rising and falling edges of **C1** are synchronized with the clock edge (with some delay included). The state of the **C1** signal at the first clock edge is low so the data is not loaded from R2 to R1. Remember, both the **C1** signal needs to be high and the rising edge of the clock has to be present in order for the load to occur. Not that the data on the **A** signal is arbitrary and the number chosen have no particular meaning. The initial value of the **B** signal is arbitrarily place in an unknown state but becomes known on the rising clock edge.



(a) (b)

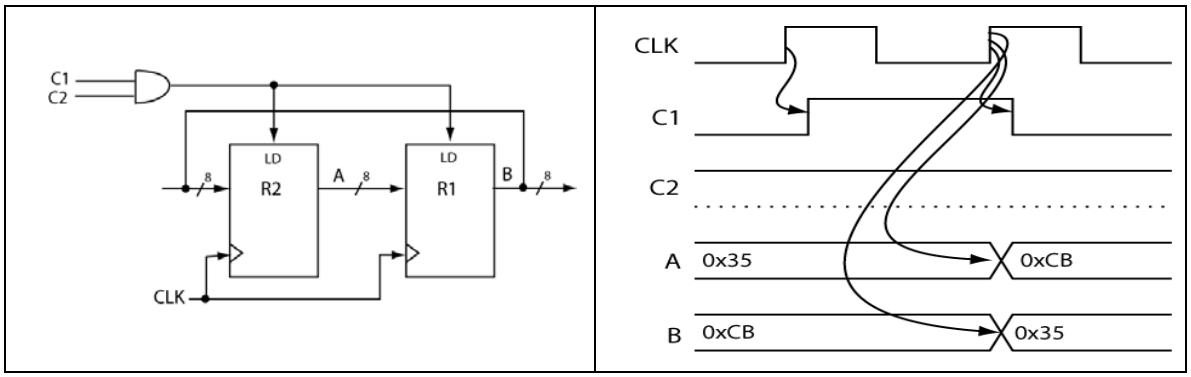
Figure 6-2: Solution and timing diagram for Example 2.

Example 6.3

Draw a circuit that implements the following RTL statement: C1,C2: R1 ← R2, R2 ← R1

Solution: This problem is slightly different from the previous problem. This type of RTL statement shows that more than one data transfer can happen simultaneously as indicated by the comma-separated equations on the right side of the colon. The left side of the colon indicates a more complex condition that allows the data transfers to happen. Figure 6-3(a) shows the circuit having this functionality. Note that the comma-separated conditions of "**C1**,**C2**" say that both **C1** *and* **C2** need to be asserted in order for the transfers to occur. The *and* in this statement can be nicely implemented as an AND gate as shown in Figure 6-3(a). Figure 6-3(b) shows an accompanying timing. The only thing this circuit does is swap the data between the R2 and R1 registers.

One other important matter to concern yourself with in Figure 6-3(b) is the relation between the CLK signal and the C1 signal. The diagram lists that the state change in the C1 signal is caused by the CLK signal. The underlying and unspoken detail here is that some other circuit in the system (that is not listed) is going to change the state of the C1 signal. In other words, the C1 signal could be considered the output of some FSM (remember the Z outputs?) that is subsequently a function of other unmentioned input. The point here is that many control signal in digital systems change because of the latest clock edge.



(a) (b)

Figure 6-3: Solution and timing diagram for Example 3.

Example 6.4

Draw a circuit that is able to implement the following RTL statements. Assume you have a standard n-bit register available to you that has a LD (load) input.

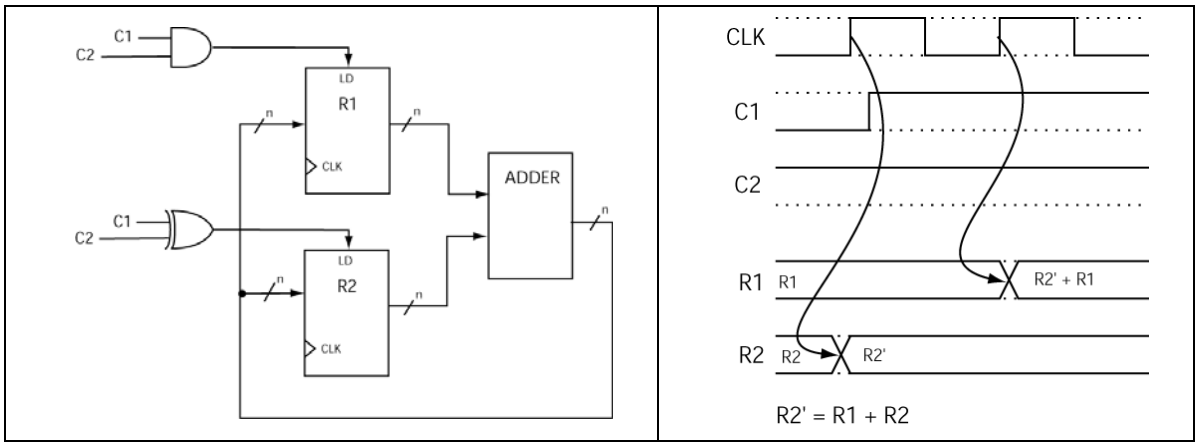
$$C1 \oplus C2 : R2 \leftarrow R1 + R2$$
$$C1, C2 : R1 \leftarrow R1 + R2$$

Solution: The best approach to take when approaching these circuits is to start listing what you know about the problem. The things that you should realize about this problem are listed below. The final circuit is shown in Figure 6-4(a).

- The "+" operator on the right side of the colon represents addition. If this operator had appeared in the left side of colon, it would have represented an OR operator. The presence of an addition operator implies that you have some hardware capable of performing the operation. In this case, the hardware is a simple *adder*, such as an RCA. The typical adder adds two n-bit numbers and outputs the results.
- The circuit will require two registers. You know this because you see that there is an R1 and an R2 but no other registers.
- The output of each register is going to be added. This means that the outputs of the registers must be the inputs to the adder.
- The result of the addition must be made available to the inputs of both the R2 and R1 register. This means the adder output is a source that has two destinations: the input of the R1 and R2 registers.
- There is some extra controlling logic required to enable the loading under the appropriate conditions. This includes the AND gate and an EXOR gate.

Figure 6-4(b) shows a timing diagram associated with the circuit solution of Figure 6-4(a). There are a few things to notice in this diagram:

- All transitions occur on the rising clock edge.
- Output data from the adder is transferred to the R2 register on the first rising clock edge (and not the second rising clock edge) because the conditions of C1 and C2 satisfy the loading logic for the register (the XOR gate).
- Output data from the adder is transferred to the R1 register when the state of signals C1 and C2 satisfy the logic for the load input of the R1 register (second rising clock edge only).



(a) (b)

Figure 6-4: The solution for Example 1 (a) and an associated timing diagram (b).

6.3 Microoperations and Data Transfers

Microoperations are elementary operations that are performed on data stored in one or more registers. The thing to note here is that it is the registers themselves that have the ability to perform elementary operations. When a register performs one of these elementary operations, it is considered to be performing a microoperation.

Although we mentioned several types of microoperations in a previous chapter, we'll introduce more microoperations in this chapter and we'll divide them into specific types. The reality is that the types listed are somewhat arbitrary. Not only do they not include every possible microoperation ever possible on any piece of hardware, some of the microoperations listed can fall into more than one of the listed types. Some of the instructions that can logically fit into more than one category will be duly noted. The following classification then is mostly for instructional purposes so don't try to read too much into it. The four major types of microoperations can be classified as follows:

- *Transfers* – data is not changed as data passes from one register to another
- *Arithmetic* – some arithmetic function is performed on data in registers
- *Logic* – some logical-type bitwise manipulation is performed on data in the registers
- *Shift* – the change in register data can be characterized by a shift in the data

6.3.1 Transfer Microoperations

Equation 6.2 shows a typical transfer microoperation represented by an RTL statement. In this equation, the contents of register R2 are transferred to register R1 under the condition that X is asserted. This transition, as are most all microoperation represented by RTL, is synchronized to some clock edge.

$$X : R1 \leftarrow R2$$

Equation 6.2: A typical transfer microoperation.

6.3.2 Arithmetic Microoperations

There are a few standard arithmetic microoperations that everyone knows and loves. Table 6.1 shows some of the more popular arithmetic microoperations. The most important thing to remember about the microoperations listed in Table 6.1 is that writing the equation means that you can either currently perform the operation (the hardware, in this case some arithmetic circuit, exists) or you'll soon be able to perform the operation (you're designing the hardware capable of performing the given function).

Arithmetic Micro-ops	Worthy Comment	Hardware Possibilities
!Cin : R1 ← R2 + R3	Addition; source registers are not changed; Assumes there is some circuitry that is capable of doing the addition (such as an adder); the values of R2 and R3 do not change.	The output of R2 and R3 is directed to the input of an adder. The output of the adder is connected to the input of R1.
!Cin : R1 ← R2 + R1	Addition; one source is destination; the value of R2 generally does not change.	The output of R2 and R1 are connected to the input of an adder. The output of the adder is connected to the input of R1.
!Cin : R3 ← R3 + R3	Addition; doubling circuit	The output of R3 is connected to both inputs of an adder. The output of the adder is connected to the input of R3.
$Cin : R2 \leftarrow R3 + \overline{R4} + 1$	Subtraction; the 2's compliment thing (R2 = R3 - R4);	The output of R3 and the complimented output of R4 connects to the input of an adder. The Cin input of the adder (generally speaking, we'll consider all adders in CPE 229 to be ripple carry adders) is set to '1' which is subsequently included in the addition.
$R5 \leftarrow \overline{R5}$	Complement contents of R5 (1's complement)	The output of R5 feeds into a row of inverters; the output of the inverters feed back to the R5 inputs.
$Cin : R5 \leftarrow \overline{R5} + 1$	Negation (multiply by -1); value is R5 is replaced by -R5	The compliment of R5 and '0' are connected to the inputs of an adder. The Cin input of the adder is set to '1'.
R1 ← R1 + 1	Increment R1; R1 register changes	The magic increment input of a counter.
Cin : R1 ← R2 + 1	Add 1 to R2 and store result in R1; the value of R2 does not change.	The output of R2 is added to 0 and the Cin input is a '1'.
R1 ← R1 – 1	Decrement R1; R1 Register changes	The standard decrement operation of a counter.
R2 ← R1, R1 + 1	Assign R1 to R2; the R1 value is then incremented.	The output of counter R1 is latched to register R2. At the same time, the value in the R1 register is incremented.

Table 6.1: Some popular arithmetic microoperations.

6.3.3 Logic Microoperations

There are a handful of logic microoperations that provide useful tools for manipulating the data in registers. Logic operations are generally considered to be bitwise in nature, meaning that the associated logic operator is applied to each of the bits in the registers on a one-to-one basis. Table 6.2 shows some of the more common logic microoperations.

Logic Micro-ops	Worthy Comment	Hardware Possibilities
$R5 \leftarrow \overline{R5}$	Logical bitwise complement (1's complement); complement the current value of R5 and return the new value to R5; The current value of R5 changes.	The output of register R5 is complimented and fed to the inputs of R5.
$R5 \leftarrow \overline{R2}$	Logical bitwise complement (1's complement); complement the current value of R2 and store the result in R5; The current value of R2 generally does not change.	The output of register is complimented and becomes the input of the R5 register.
R0 ← R1 AND R2	Logical bitwise AND of R1 and R2; the result is stored in R0; the current values of R1 and R2 generally do not change.	The output of R1 is ANDed with the output of R2; the result becomes the input to R0.
R1 ← R1 AND R2	Logical bitwise AND of R1 and R2; the result is stored in R1; the current value of R2 generally does not change.	The output of R1 is ANDed with the output of R2; the result becomes the input to R1.
R3 ← R1 OR R2	Logical bitwise OR of R1 and R2; the result is stored in R3; the current values of R1 and R2 generally do not change.	The output of R1 is ORed with the output of R2; the result becomes the input to R3.
R1 ← R1 XOR R2	Logical bitwise Exclusive OR of R1 and R2; the result is stored in R1; the current value of R2 generally does not change.	The output of R1 is EXORed with the output of R2; the result becomes the input to R1.

Table 6.2: Some popular logic microoperations.

6.3.4 Shift Microoperations

Here is the list of basic shift-type operations:

1. **Simple shifts**: The simple shift would include single shifts in either the left or the right direction. This shift is referred to as simple because the shifts that follow are somewhat less simple.

2. **Rotates**: The rotate operations (rotate left and rotate right) either feeds the MSB to the LSB (on a left shift operation) or the LSB to the MSB (on a right shift operation). All other bits shift accordingly.

3. **Arithmetic shifts**: The arithmetic shift is for operations where the bits stored in the register are considered to be a signed number. In this case, the MSB is considered the sign bit and its present state must be preserved in both the left and right shift operations.

4. **Barrel shifts**: A barrel shift essentially performs more than one simple shift (in any one direction) in a single clock cycle. The distance of the barrel shift is arbitrary but is indicated in the RTL equation with the "Xx" notation (where the capital X represents the effective number of bit shifts). These shifts

are actually quite useful since they provide a fast multiplication and division (depending on shift direction). The only catch here is that the divisions and multiplications need to be by a factor of two. The barrel shift can be used to instantly scale a mathematical result thus saving clock cycles that you would need to expend to do the shifts (multiplication or division) on separate clock cycles.

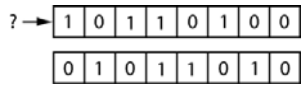
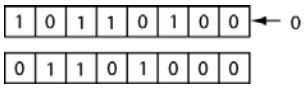
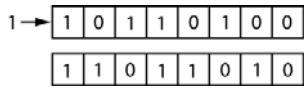
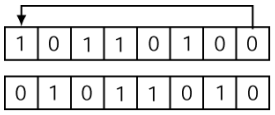
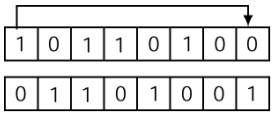
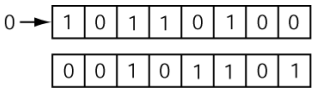
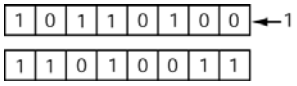
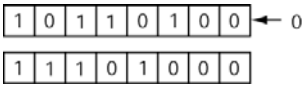
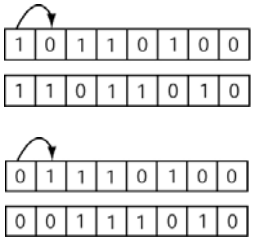
Shift Micro-ops	Worthy Comment	Worthy Picture
R0 ← sr R0	shift right of R0; result is store in R0; some undetermined valued is feed in to the left side of the register.	
R2 ← sl R2 (r-0)	shift left of R2; feed in '0' from right side	
R2 ← sr R2 (l-1)	shift right of R2; result stored in R2; feed in '1' from left side	
R2 ← rr R2	rotate right; the LSB is transferred to the MSB;	
R2 ← rl R2	rotate left; the MSB is transferred to the LSB.	
R2 ← bsr2x R2 (l-0)	barrel shift right 2x (two simple shifts); result stored in R2; feed in '0' from left	
R3 ← bsl2x R3 (r-1)	barrel shift left 2x; result stored in R3; feed in '1's from right. This would be the same as two simple shift lefts that fed a 1 into the right.	
R4 ← asl R4 (r-0)	arithmetic shift left; sign bit is copied from left side with each shift; '0' is fed into the right side of the register; this is essentially a multiply by two on a signed number.	
R5 ← asr R5	arithmetic shift right; sign bit is not altered any shift; the sign bit is copied from the MSB to the MSB -1 on each right-shift; this is essentially a divide by two on a signed number.	

Table 6.3: Some popular shift-type microoperations.

One important thing to notice about the RTL equations written above is that none of them contain conditions. Generally speaking, there will be some unit in your computer that handles all of these functions. The way you

would officially tell the unit to perform a given function is to tweak the proper control signals (such as "select-type" signals). These control signal values should appear in the RTL statements above. The above equations do not because our discussion was primarily an introduction.

The last comment on the RTL matter is fact that only conditions appear on the left side of the colon. You need to remember this because the "+" operator sometimes represents a logical OR and at other times represents addition. An OR operation is considered a condition and can appear on the left side of the colon. However, an addition operation could not be construed as a condition and would never appear on the left side of the colon. Therefore, a "+" operator has special context in RTL equations. For example, in the equation listed in Equation 6.3, the "+" operator on the left side of the colon represents an OR operation while the "+" operator on the right side of the colon represents an addition operator. If ever in doubt, feel free to spell it out absolutely clearly in written English, with footnotes, or with arrows.

$$K1 + K2 : R1 \leftarrow R2 + R3$$

Equation 6.3: Equation showing "+" operator but having two different meanings.

A good approach to truly understanding the various shift-type microoperations is to implement a circuit that can perform the operations. A great way to describe a digital circuit is to model it using VHDL. The justification for spending time at this level is that somewhere in the datapath of most computers is a shifting unit. The generic form of the shifting unit is a black box as shown in Figure 6-5. As you'll see, these units can be modeled using VHDL in a relatively painless manner. Check out this following example.

Example 6.5

Design a synchronous shifting unit with the following characteristics:

a. 8-bit output (D)
b. LD input asynchronously transfers 8-bit D_LOAD value to D
c. 3-bit CTRL signal synchronously causes following functions:
    1. sr (l-0) : shift right; feed in '0' from left side
    2. sl (r-1) : shift left; feed in '1' from the right side
    3. rr (rotate right)
    4. rl (rotate left)
    5. swap nibbles (swap lower bits with upper 4-bits; 4-bits = nibble)
    6. swap individual bits (swap MSB with LSB, swap MSB-1 with LSB +1, etc.)
    7. multiply by 4 (2x left barrel shift; unsigned)
    8. divide by 4 (2x right barrel shift; unsigned)

Solution: The best place to start is with a high-level black box diagram. It happens that Figure 6-5 shows such a diagram. The information in Figure 6-5 comes from the problem description.

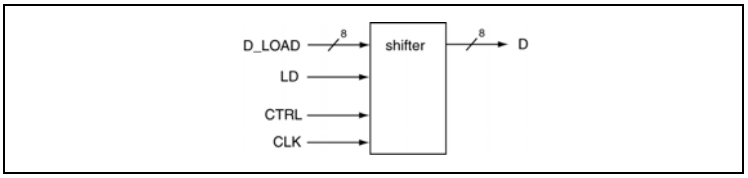


Figure 6-5: Black box for a shifter unit.

```
entity shifter is
    Port (  CLK,LD : in std_logic;
              CTRL : in std_logic_vector(2 downto 0);
            D_LOAD : in std_logic_vector(7 downto 0);
                 D : out std_logic_vector(7 downto 0));
end shifter;

architecture my_shifter of shifter is
   signal d_t : std_logic_vector(7 downto 0);
begin
   process (CLK,CTRL,D,LD,D_LOAD)
      begin
      -- asynchronous load
      if (LD = '1') then
         d_t <= D_LOAD;
      elsif (rising_edge(CLK)) then
         case CTRL is
            -- sr (l-0)  (shift right feed in '0' on left)
            when "000" =>  d_t <= '0' & d_t(7 downto 1);

            -- sl (r-1)  (shift left feed in '1' on right)
            when "001" =>  d_t <= d_t(6 downto 0) & '1';

            -- rr (rotate right)
            when "010" =>  d_t <= d_t(0) & d_t(7 downto 1);

            -- rl (rotate left)
            when "011" =>  d_t <= d_t(6 downto 0) & d_t(7);

            -- nibble swap (nibble is 1/2 a byte or 4-bits)
            when "100" =>  d_t <= d_t(3 downto 0) & d_t(7 downto 4);

            -- bit swap
            when "101" =>  d_t <= d_t(0) & d_t(1) & d_t(2) & d_t(3) &
                                  d_t(4) & d_t(5) & d_t(6) & d_t(7);

            -- multiply by 4 (2x left barrel shift)
            when "110" =>  d_t <= d_t(5 downto 0) & "00";

            -- divide by 4 (2x right barrel shift)
            when "111" =>  d_t <= "00" & d_t(7 downto 2);

            when others => d_t <= (others => '0'); -- don't get here!
         end case;
      end if;
   end process;
   D <= d_t;
end my_shifter;
```

Figure 6-6: The VHDL code for the shifter circuit.

6.4 Data Transfer Circuits

As you can tell by now, a functional datapath passes data around in a useful manner. We need to get into some of the specifics of how the data is passed around; that is, we need to look at the underlying hardware and understand exactly how things are done so that we can orchestrate such transfers. There are roughly four different circuit styles for transferring data around:

1) MUX-based transfers
2) bus-based transfers
3) tri-state bus-type transfers

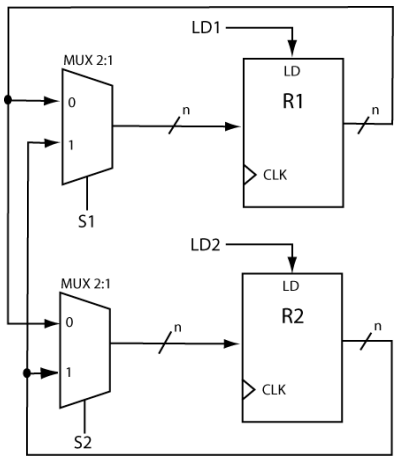
4) open collector

6.4.1 MUX-Based Data Transfers

Figure 6-7(a) shows a typical circuit that performs MUX-based data transfers. The table in Figure 6-7(b) includes some example microoperations and the control signals required to perform those operations. We use the **Sx** signals to control the two MUXes and the **LDx** signals are used to control the loading of data into the various registers. The clock signal is omitted in an effort to make the diagram look less trashy than it currently appears. The width of the bus for this example and the other examples that follow are assumed to be of generic width "n". All transfers are synchronized on the rising edge of the clock. Below are a few other things to note about this circuit.

- If a register does not need to be loaded for a particular microoperation, the LD signal is held low. The state of the corresponding MUX control signal is therefore a "don't care" but is listed as '0'.
- Most often, conditions of "don't care" are not included in the RTL statement. In general, for a given RTL statement, you should specify all associated signals to leave no room for testy ambiguity.
- Each signal source contains one and only one destination.
- The fact that the data signals in this example are of "width n" implies that they are bundles. In computerland, the word bus is an overused and ambiguous term; the word bundle is a better term.

And alas, Table 6.4 shows the bit control information in RTL form.



(a)

Microoperation	S1	S2	LD1	LD2
R2 ← R1	0	0	0	1
R1 ← R2	1	0	1	0
R2 ← R1, R1 ← R2	1	0	1	1
R2 ← R1, R1 ← R1	0	0	1	1

(b)

Figure 6-7: A circuit for MUX-based transfers (a) and control signals necessary to perform the listed microoperations (b).

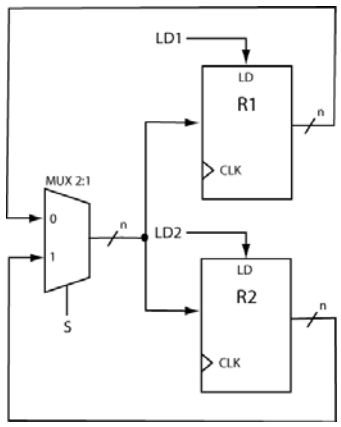
Microoperation	S1	S2	LD1	LD2	RTL
R2 ← R1	0	0	0	1	$\overline{S2}, \overline{LD1}, LD2 : R2 \leftarrow R1$
R1 ← R2	1	0	1	0	$S1, LD1, \overline{LD2} : R1 \leftarrow R2$
R2 ← R1, R1 ← R2	1	0	1	1	$S1, \overline{S2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R2$
R2 ← R1, R1 ← R1	0	0	1	1	$\overline{S1}, \overline{S2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R1$

Table 6.4: The table from Figure 6-7(b) with associated RTL statements.

6.4.2 Bus-Based Data Transfers

Although MUX-based transfers are friendly and versatile, they are considered a waste of hardware. The versatility comes from the fact that you can perform just about any action you can dream up but it comes as the cost of extra hardware. Bus-based transfers are similar to MUX-based transfers but are not quite as versatile. Figure 6-8(a) shows a circuit for bus-based transfers. The microoperations in Figure 6-8(b) are the same ones listed in Figure 6-8(a). Here are a few things to note about this circuit:

- This bus-based circuit has less hardware than the MUX-based circuit. This ends up being a trade-off with functionality as is noted in the next bulleted item.
- Due to the limited hardware connections (compared to the MUX-based transfers), one of the desired microoperations cannot be done. Bummer!
- This is called a bus-based transfer because there is one bus that had one source but multiple destinations. Note that in the MUX-based transfers, each source had only one destination.



(a)

Microoperation	S	LD1	LD2
R2 ← R1	0	0	1
R1 ← R2	1	1	0
R2 ← R1, R1 ← R2	Can't be done		
R2 ← R1, R1 ← R1	0	1	1

(b)

Figure 6-8: A circuit used for bus-based transfers (a); control signals to perform microoperations (b).

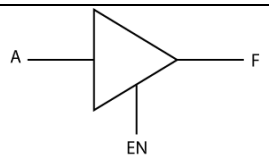
Microoperation	S	LD1	LD2	RTL Statement
R2 ← R1	0	0	1	$\overline{S}, \overline{LD1}, LD2 : R2 \leftarrow R1$
R1 ← R2	1	1	0	$S, LD1, \overline{LD2} : R1 \leftarrow R2$
R2 ← R1, R1 ← R2	Can't be done			bummer!
R2 ← R1, R1 ← R1	0	1	1	$\overline{S}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R1$

Table 6.5: The table from Figure 6-8(b) with added RTL statements.

6.4.3 Tri-State Bus-Based Transfers

These transfers are centered about the use of a tri-state buffer as is shown in Figure 6-9(a). The name tri-state comes from the fact that the output of the buffer can have three possible states as is shown in Figure 6-9(b). Two of the three states are the now infamous 1's and 0's while the other state is the *high-impedance* state signified with the letter Z. When the circuit goes into the high-impedance state, no current flows through the device. Any time there is not current flowing through a conductive path, the path is considered an open circuit. In this case, if there is an open circuit, the device is effectively removed from the circuit. A better wording for this would be that the device has no significant effect on the circuit since no one is physically removing the device from the circuit. The EN (enable) input essentially enables the input to appear on the output of the device as in indicated with the truth table and compressed truth table of Figure 6-9(b) and Figure 6-9(c), respectively.

The hearts of tri-state bus transfers are registers that contains tri-state buffers on the output of the devices. In other words, each of the bits stored in the register contains its own tri-state buffer. The EN input is connected to each of the tri-state buffers in the register and controls each of the output bits in parallel. As is shown in the circuit of Figure 6-10(a), we indicate registers with tri-state outputs with the triangles on the outputs.



(a)

EN	A	F
0	-	Z
1	0	0
1	1	1

(b)

EN	F
0	Z
1	A

(c)

Figure 6-9: A tri-state buffer (a) and associated truth tables in full and compressed form (b) and (c).

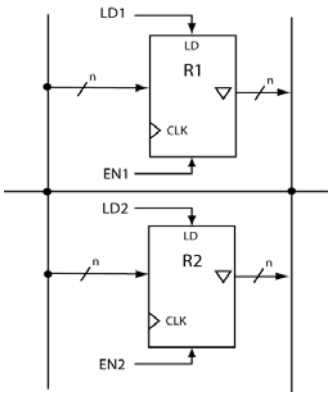
The circuit in Figure 6-10(a) has one major difference from the two previous circuits. The differences between each the three types of transfers we're looking at are highlighted in Table 6.6. The fact that tri-state buses generally have more than one source means that there is possibility of *bus contention*. Bus contention occurs when two more output devices (registers in this case) drive their stored data onto the bus line at the same time. This results in indeterminate circuit behavior, so you should definitely avoid it. For example, if one output device drives the bus with all 1's and another drives the bus with all 0's, what would some input device see on these lines?

The way to avoid bus contention is to make sure that no more than one output device is driving the bus lines at one time. The way to drive the bus is to assert the EN input on the registers so only one of these should be asserted at any one time. When the device is not asserted, the device is essentially removed from the circuit (although the inputs of the device are generally able to latch data).

Transfer Type	Interesting Bus Characteristic for Buses
MUX-based	one source - one destination
Bus-based	one source - multiple destinations
Tri-state bus-based	multiple sources - multiple destinations

Table 6.6: The major differences between transfer types.

Figure 6-10(a) shows the resulting circuit. The microoperations listed in Figure 6-10(b) are the same microoperations for the previous types of data transfers. Once again, as you can see from the circuit diagram of Figure 6-10(a), there seems to be less hardware in the circuit as compared to MUX-based and bus-based transfers. As is shown in Figure 6-10(b), one of the RTL statements is still not possible. The most important thing to note from the table in Figure 6-10(b) is the fact that for any given RTL statement, only one of the register enables is active at a time. If more than one enable signal was active on a given bus line, there would be bus connection



(a)

Microoperation	EN1	EN2	LD1	LD2
R2 ← R1	1	0	0	1
R1 ← R2	0	1	1	0
R2 ← R1, R1 ← R2	Can't be done			
R2 ← R1, R1 ← R1	1	0	1	1

(b)

Figure 6-10: A circuit for tri-state bus-based transfers (a); signals controlling the microoperations (b).

Microoperation	EN 1	EN 2	LD 1	LD 2	RTL Statement
R2 ← R1	1	0	0	1	$EN1, \overline{EN2}, LD1, \overline{LD2} : R2 \leftarrow R1$
R1 ← R2	0	1	1	0	$\overline{EN1}, EN2, LD1, \overline{LD2} : R2 \leftarrow R1$
R2 ← R1, R1 ← R2	Can't be done				unkempt
R2 ← R1, R1 ← R1	1	0	1	1	$EN1, \overline{EN2}, LD1, LD2 : R2 \leftarrow R1, R1 \leftarrow R1$

Table 6.7: The table from Figure 6-10(b), with associated RTL statements.

And finally, there is an alternate method that is commonly used to draw the circuit of Figure 6-10(a). A somewhat shorthand notation for the tri-state bus transfer circuit of Figure 6-10(a) is shown in Figure 6-11. These two circuits are equivalent but note that the circuit of Figure 6-11 is much nicer to look at. The double arrows on the bus lines indicate that the lines are both inputs and outputs.

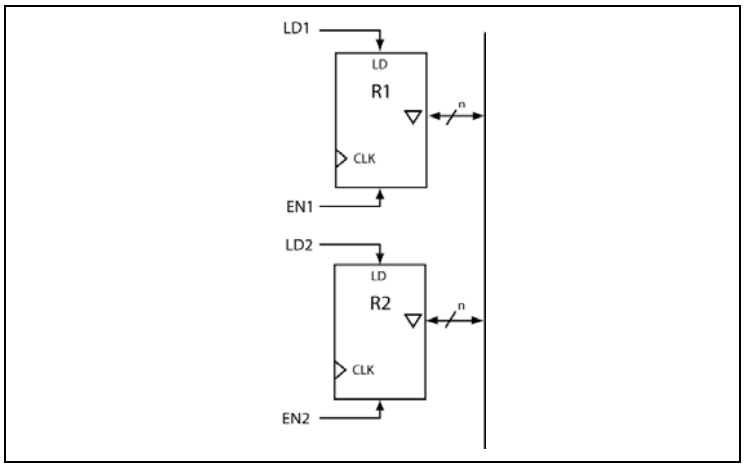


Figure 6-11: An alternative method to draw the circuit shown in Figure 6-10(a).

Example 6.6

Using the circuit shown in Figure 6-12(a), write the RTL equations that would accomplish the following list two sets of tasks: (write two different equations).

1) Transfer R1 to R2; increment R1
2) Transfer R2 to R1

Solution: Once again, there are a few quick things to notice about this circuit:

- Each of the registers has tri-stated outputs. This requires that the registers have enable signals, which must be asserted in order to drive that register's data on to the bus. This also means that only one of the enable signals (EN1 and EN2) better be asserted at one time.

- The R1 register has a CNT_EN input, which roughly stands for count enable. This implies that the R1 register is a counter. Looking at the first required transfer indicates an increment operation (Rx ← Rx + 1) which, by using the logic of the previous problem, requires an adder. However, since this register is a counter and counters typically count up one value at a time in a synchronous fashion, all you need to do is assert the count enable to induce the required increment operation. In this case, assume that if the count is not asserted, the value stored in the register does not change.

Here are the required RTL equations:

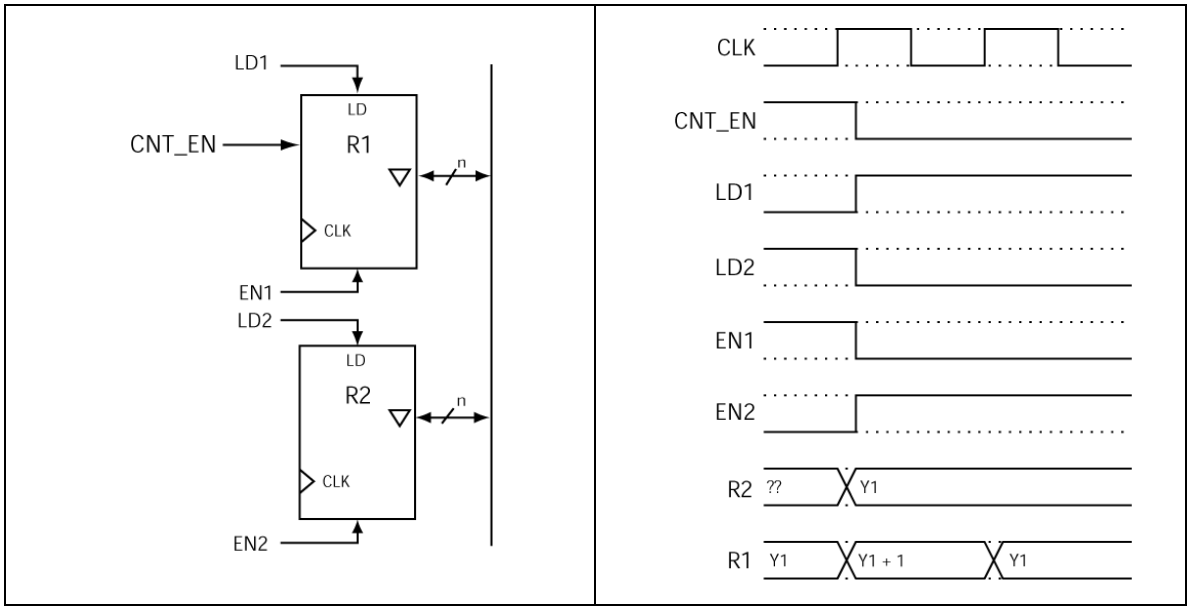
$$\text{CNT\_EN}, \overline{\text{LD1}}, \text{LD2}, \text{EN1}, \overline{\text{EN2}} : \text{R2} \leftarrow \text{R1},\ \text{R1} \leftarrow \text{R1} + 1$$

$$\overline{\text{CNT\_EN}}, \text{LD1}, \overline{\text{LD2}}, \overline{\text{EN1}}, \text{EN2} : \text{R1} \leftarrow \text{R2}$$

Figure 6-12(b) shows the associated timing diagram. One important thing to notice about this timing diagram are the transfers that occur on the first rising clock edge. On that clock edge, the data in the R1 register transfers to the R2 register; at the same time, the data in the R1 register increments. Keep in mind that these diagrams represent actual circuits. At the instance of the rising clock edge, the data transfers from R1 to R2.

Since CNT_EN is asserted, the increment of the count is also initiated on the clock edge but its effect does not happen in time for the incremented data to be transferred to the R2 register.

The above increment operation is typical in digital circuits. It's particularly important in basic computer circuits because a counter is used to "sequentially step through a stored program". Generally speaking, the output of the counter is used as an address to access an instruction in instruction memory. Once one instruction is read, the counter is incremented and then points at the next instruction in memory. We'll be looking at this in more detail in a later set of notes.



(a) (b)

Figure 6-12: The circuit for Example 3 (a), and an associated timing diagram (b).

6.5 Chapter Summary

- Register transfer notation (RTN) provides a shorthand approach to both designing and describing circuits. RTN does not have absolute standards; each RTN approach may be different from other RTNs. RTN represents a continued abstraction to higher levels of design in order to facilitate designing and understanding relatively complex circuits.

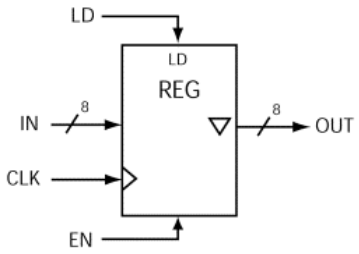
- RTN generally deals with the transfer of data from one register (the source register) to another register (the destination register). T

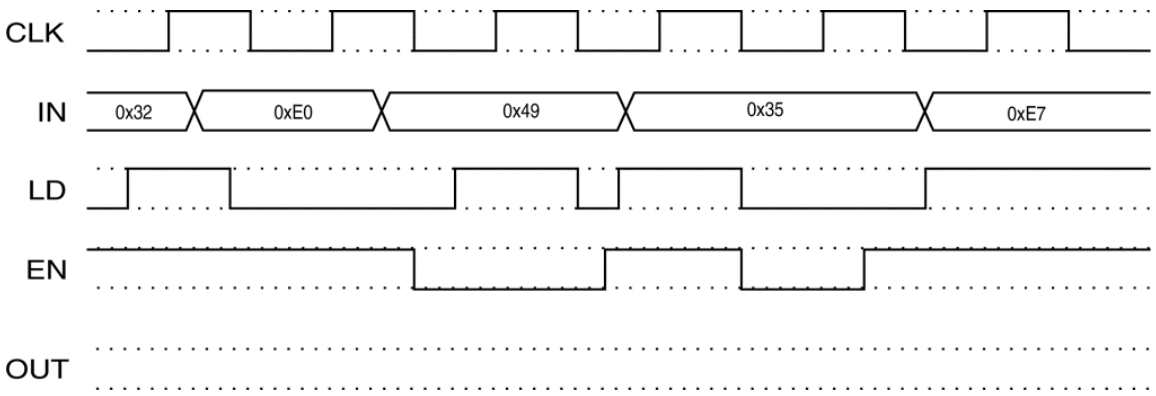
- Microoperations are elementary operations that are performed on data stored in one or more registers. We can describe the operation of many sequential circuits in terms of the various microoperations they are able to perform. There are many types of microoperations, but we generally attempt to categorize in order to support understanding their functions. Some of the more popular types of microoperations include transfers, arithmetic, logic, and shift operations.

- The key of a working computer is the ability to transfer data from a source to a destination (generally register to register, register to circuit, or circuit to register). We generally attempt to classify type of data transfers in order to support our understanding of them. The most common transfer circuits include MUX-based, bus-based, and tri-state-based circuits. Each of the circuits has their advantages and disadvantages.

6.6 Chapter Exercises

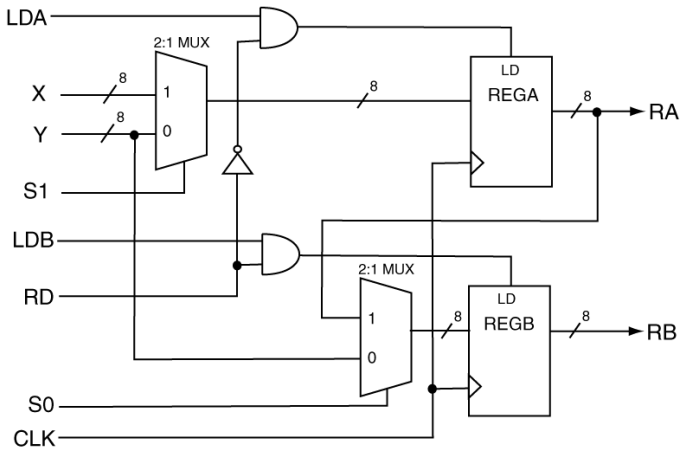
1) Use the following tri-state register diagram to complete the following timing diagram. Assume the initial value of the OUT signal is 0xA4.





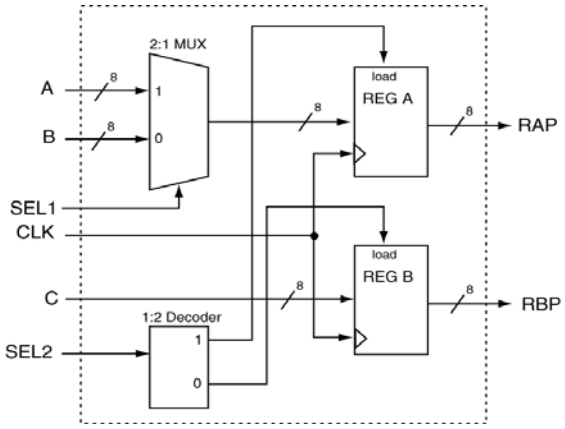
2) For this problem, complete the following two tasks:

- Write the minimum number of RTL statements that will transfer X to REGB and Y to REGA
- Write a VHDL model that could be used to synthesize following circuit



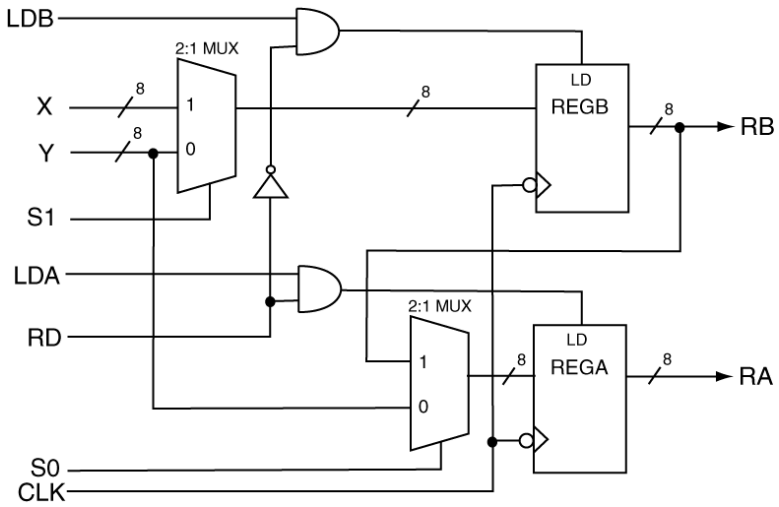
3) For this problem, complete the following two tasks:

- Write the minimum number of RTL statements that place B into REGA and C into REGB
- Write a VHDL model that could be used to synthesize following circuit



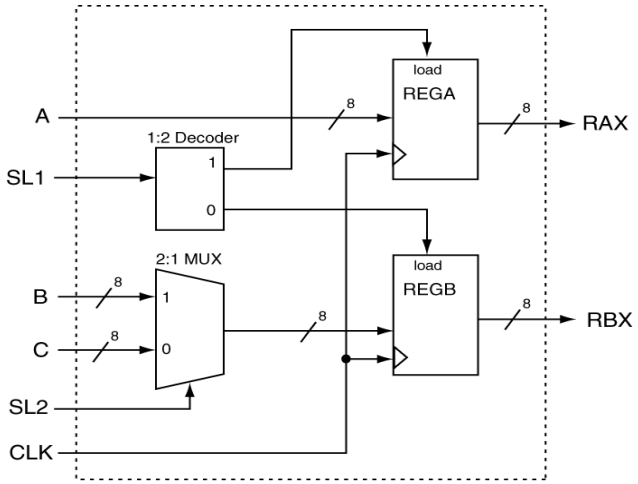
4) For this problem, do the following two tasks:

- Write the minimum number of RTL statements that will transfer X to REGA and Y to REGB.
- Write a VHDL architecture that implements the following circuit. It is not necessary to use VHDL structural modeling for your architecture.



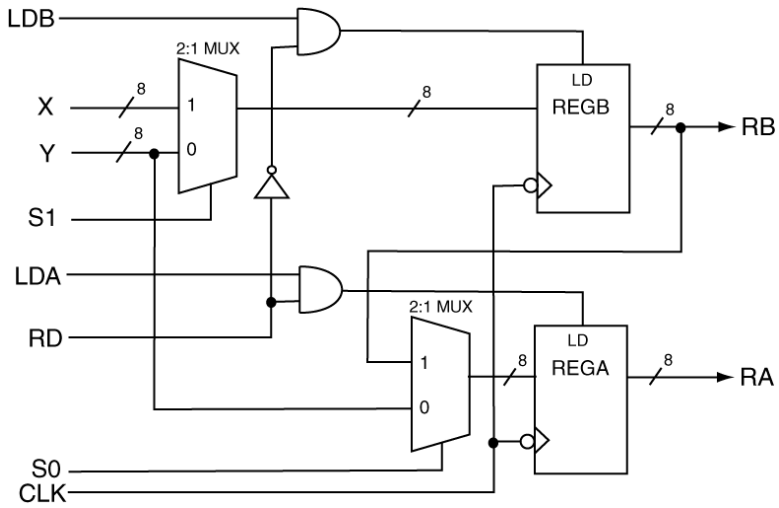
5) For this problem, do the following two tasks:

- Write the minimum number of RTL statements that will transfer A to REGA and C to REGB.
- Write a VHDL architecture that implements the following circuit.



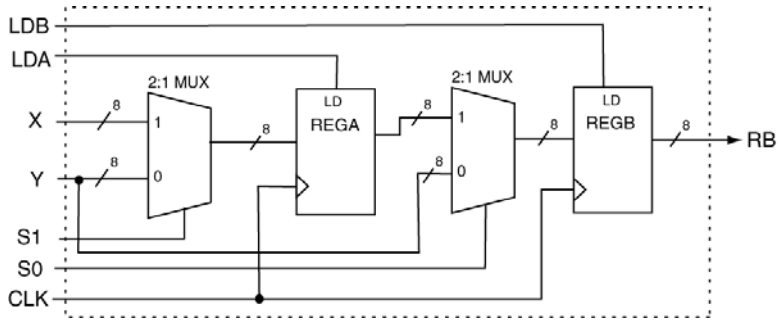
6) For this problem, do the following two tasks:

- Write the minimum number of RTL statements that will transfer Y to REGB and X to REGA
- Write a VHDL architecture that models the following circuit.



7) For this problem, do the following two tasks:

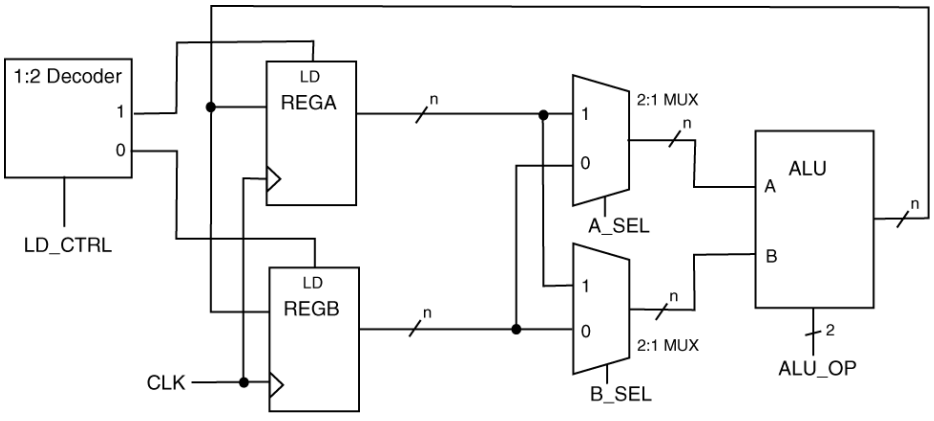
- Write the minimum number of RTL statements that will transfer X to REGA and Y to REGB.
- Write a VHDL architecture that implements the following circuit. It is not necessary to use VHDL structural modeling for your architecture.

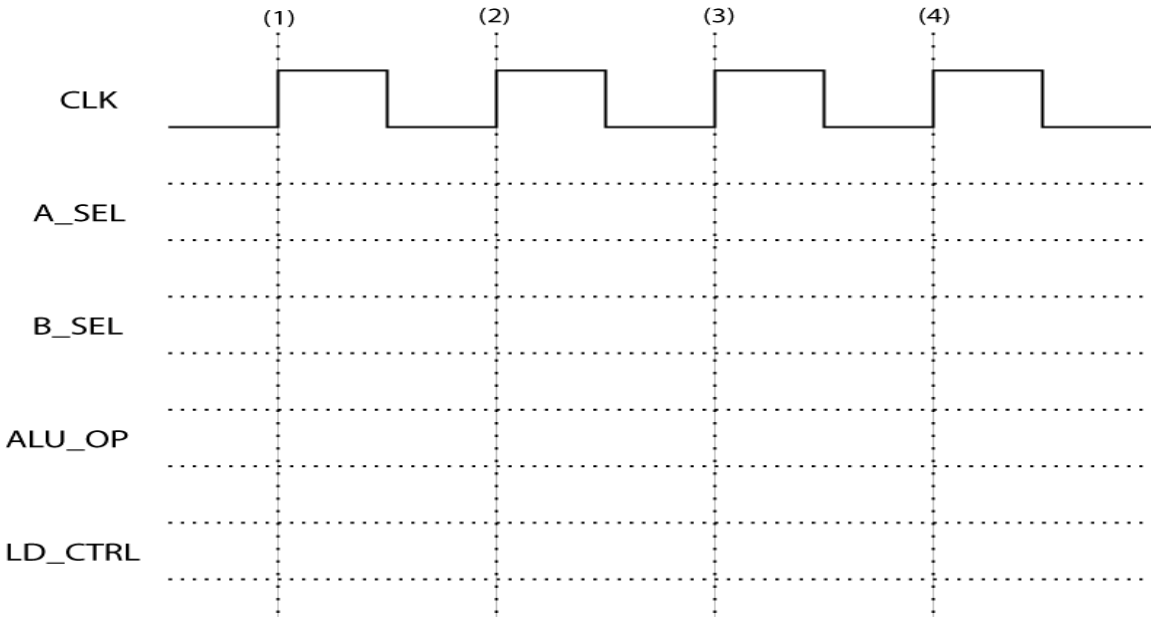


8) For this problem, provide the proper control signals that would allow the following RTL statements to occur. Complete the timing diagram below based on your provided control signals.

clock cycle	RTL
1	RA ← RA AND RB
2	RA ← RA + RA ; addition
3	RB ← RA - RB
4	RA ← rr RB ; rotate right B

ALU_OP	Operation
00	logical AND of A & B
01	rotate right B input
10	subtract B from A
11	add B to A




(1)
(2)
(3)
(4)
CLK
A_SEL
B_SEL
ALU_OP
LD_CTRL

9) The following diagram shows a circuit that can perform a mathematical operation. The two tables below describe the comparator and the Universal Shift Register (USR). Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.*

- If a GO signal is received (GO is not shown in diagram), one of the following two operations occur:
  - If (OP_A * 4) ≥ (OP_B * 4) then: REG_A ← (OP_A * 4); REG_B ← (OP_B * 4);
  - otherwise: REG_B ← (OP_A * 4); REG_A ← (OP_B * 4);

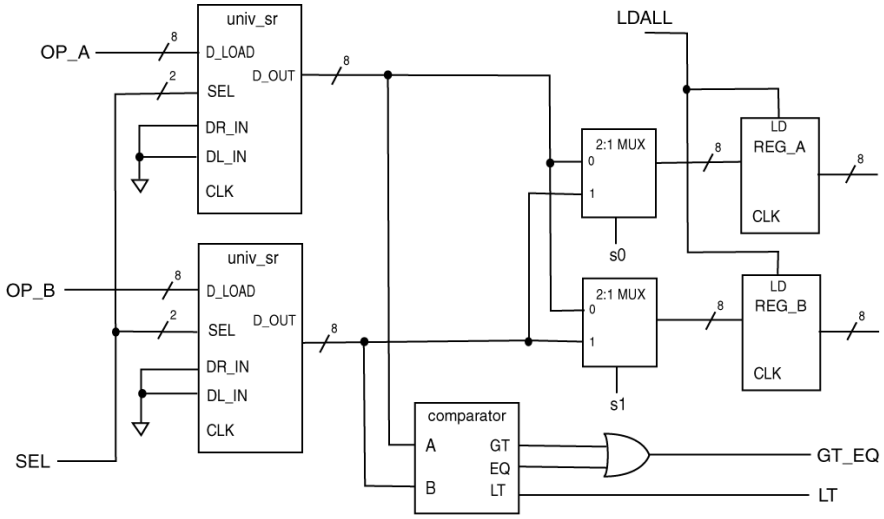
Comparator description

```
if (A > B) then GT <=
'1';
else GT <= '0';

if (A = B) then EQ <=
'1';
else EQ <= '0';

if (A < B) then LT <=
'1';
else LT <= '0';
```

Assumptions:

- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All shift register operations are synchronous
- Registers (non-USR) have synchronous load inputs (LD)

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left

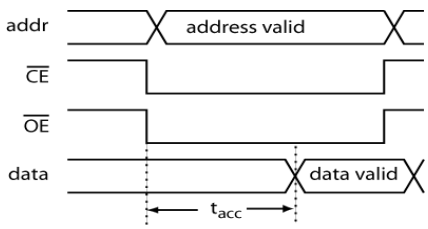


10) The following diagram shows a circuit that can perform a mathematical operation. The information below describes the Universal Shift Register (USR) and memory timing. Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.* Assume a 100MHz clock (10ns period) and a memory access time (t_{acc}) of 25ns.

❋ If a GO signal is received (GO is not shown in diagram), the following operation occurs:

- DOUT ← (B_DAT) ÷ 8

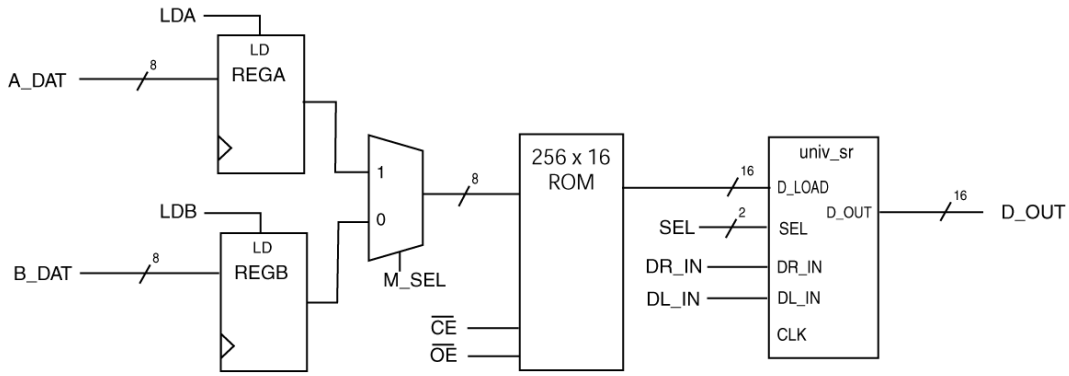
Memory Timing



Assumptions:

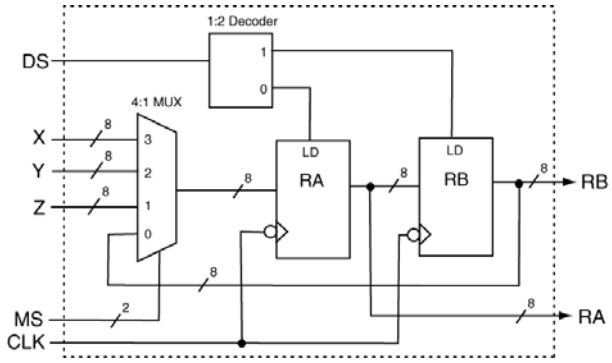
- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All shift register operations are synchronous
- Registers (non-USR) have synchronous load inputs (LD)

Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



11) For this problem, perform the following two tasks:

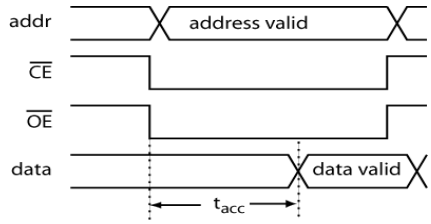
- Write the minimum number of RTL statements that will transfer Z to RB and Y to RA.
- Write a VHDL architecture that implements the following circuit.



12) The following diagram shows a circuit that can perform a mathematical operation. The information below describes the Universal Shift Register (USR) and memory timing. Provide a state diagram that could be used to control the circuit such that it performs the operation listed below. *Minimize the number of states you use in your solution.* Assume a 100MHz clock (10ns period) and a memory access time (t_{acc}) of 25ns.

- If a GO signal is received (GO is not shown in diagram), the following operation occurs:
  - DOUT ← (B_DAT) ÷ 8

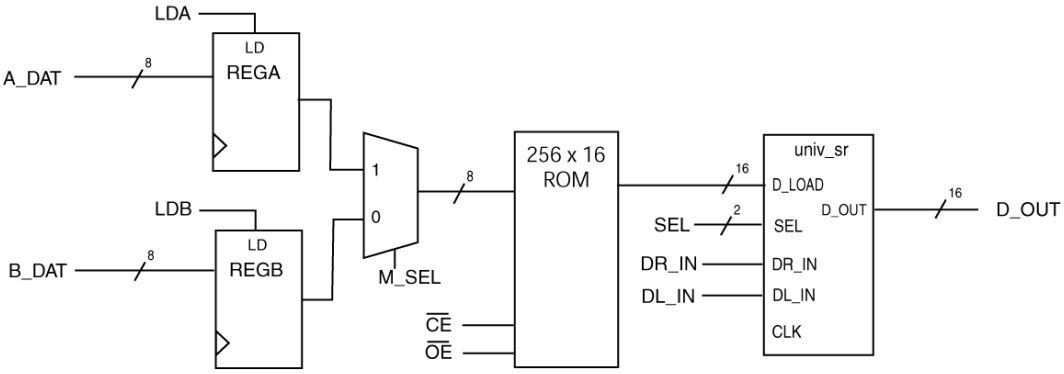
Memory Timing



Assumptions:

- DR_IN = right side input to shift register
- DL_IN = left side input to shift register
- CLK signals are connected
- All setup and hold times are met
- All shift register operations are synchronous
- Registers (non-USR) have synchronous load inputs (LD)

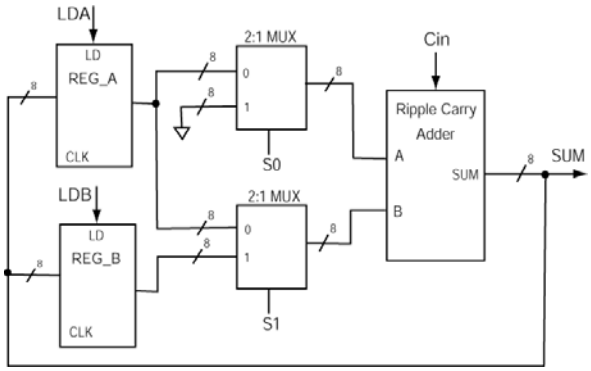
Shift Register Controls	
SEL	**Operation**
0 0	hold
0 1	parallel load
1 0	shift right
1 1	shift left



13) For this problem, perform the following two tasks:

- Write the minimum number of RTL statements that will:
  i.increment REG_B and store in REG_A
  ii.add REG_B to REG_B and store in REG_A

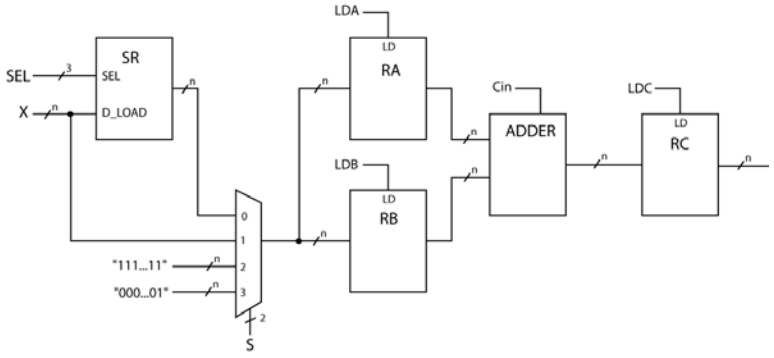
- Write a VHDL model that could be used to synthesize the following circuit.



14) For the following problem, assume the SEL inputs to the shift register (SR) cause the following operation in the SR. Assume ”SR” refers to a shift register while “sr” refers to a shift right operation.

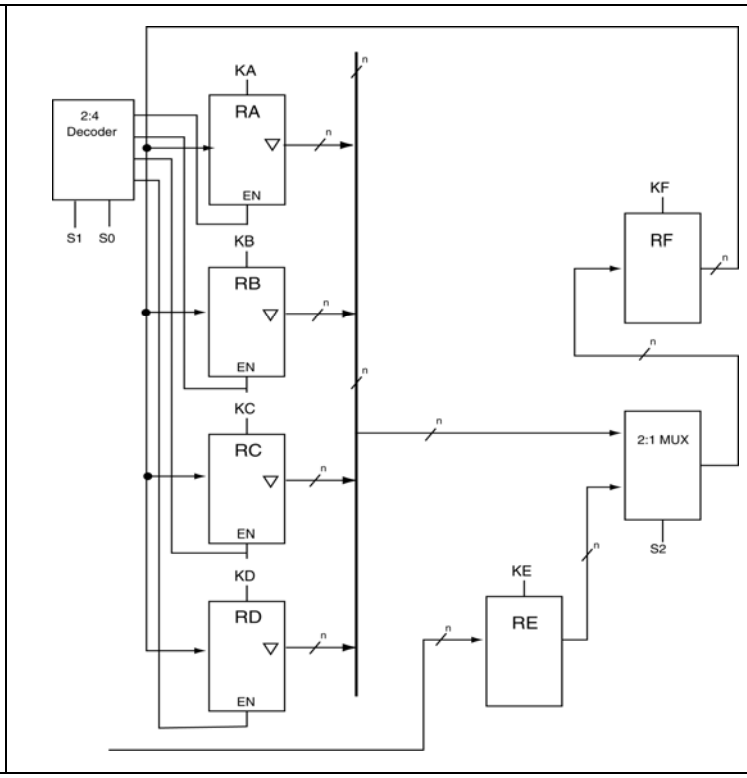
SEL	RTL Operation
000	SR ← D_LOAD
001	SR ← bsl2x SR (r-0)
010	SR ← bsr2x SR (l-0)
011	SR ← sr SR (l-0)
100	SR ←asr SR
101	SR ←asl SR (r-0)
110	SR ← others => ‘0’ (loads all zero’s)
111	SR ← SR; (hold)

(a) transfer 5X into RC (unsigned)
(b) transfer 1.5X into RC (unsigned)
(c) transfer 2X into RC (signed)
(d) transfer 0.625X into RC (unsigned)
(e) transfer 4X+2 into RC
(f) transfer 2X-1 into RC

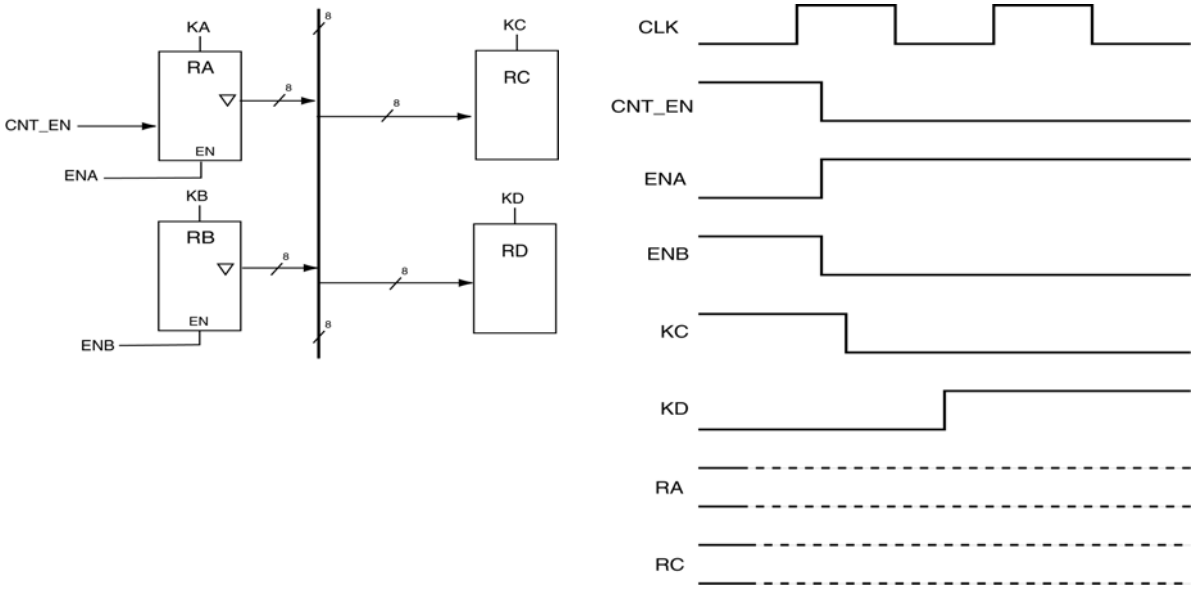


15) Write the minimum of RTL statements that will implement the following data transfers. Assume the upper-most values on the MUX and decoder start with 0 and work down to 1 and 3, respectively.

a) Transfer RE to RD
b) Transfer RB to RC
c) Transfer RA to RC as well as RA to RD
d) Transfer RD to RC and RB to RA
e) Design a FSM that could implement the set of transfers listed in a), b), c), and d).

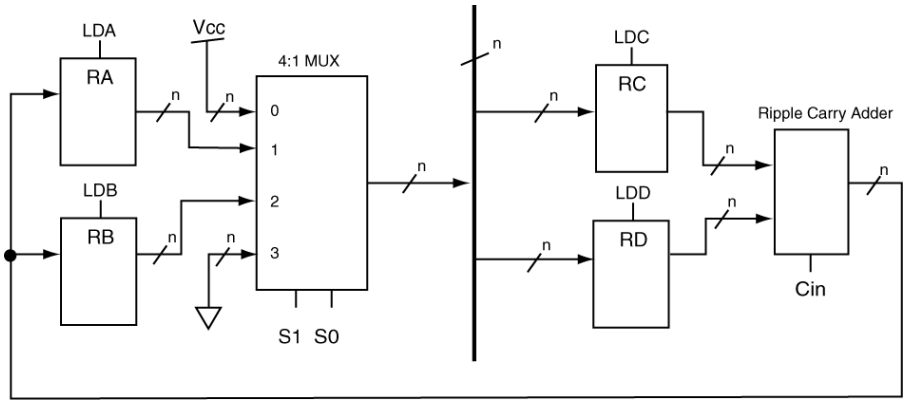


16) Use the following circuit to complete the accompanying timing diagram. Register RA is a counter with a count enable input. The initial value (in hex) on each register is RA=AA, RB=BB, RC=CC, and RD=DD. Consider the rising edge of the clock to be the active edge.



17) Using the circuit provided, write the minimum number of RTL statements that will implement the following data transfer. Consider the circuit elements with Rx labels to be registers with load inputs LDx (listed) as well as clock inputs (not listed). Data is loaded into the registers only on the active clock edge. The clock signal is not shown.

- Increment the value in RB and load the result into RA
- Add RA to RB and store the result in both RA and RB
- Decrement the value in RB and store the result in RB
- Transfer the value in RA to RB
- Clear RA and set RB (make all the bits 0 and 1, respectively)
- Load the -1 into RA
- Design a FSM that would decrease the value in RB by three and store the result in RA.



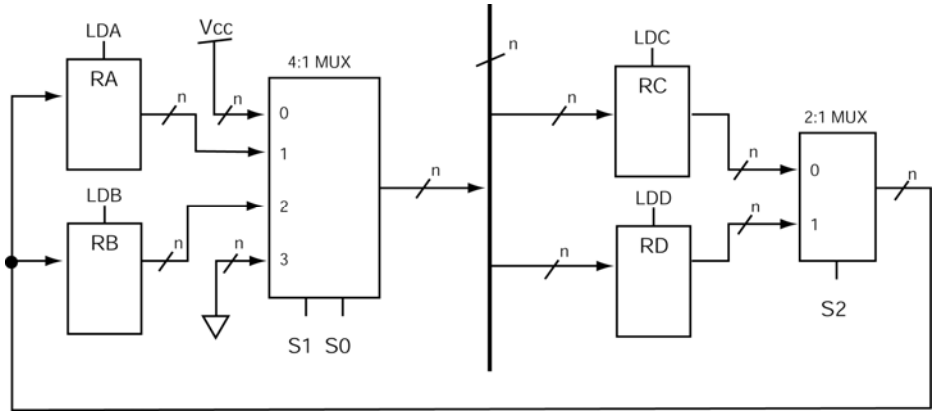
18) Using the circuit provided, write the minimum of RTL statements that will implement each of the following data transfers. Consider the circuit elements with Rx labels to be registers with load inputs LDx (listed) as well as clock inputs (not listed). Data is loaded into the registers only on the active clock edge. The clock signal is not shown.

A)

- Sets RD
- Transfer RC to RD

B)

- Clear RC
- Transfer RA to RB

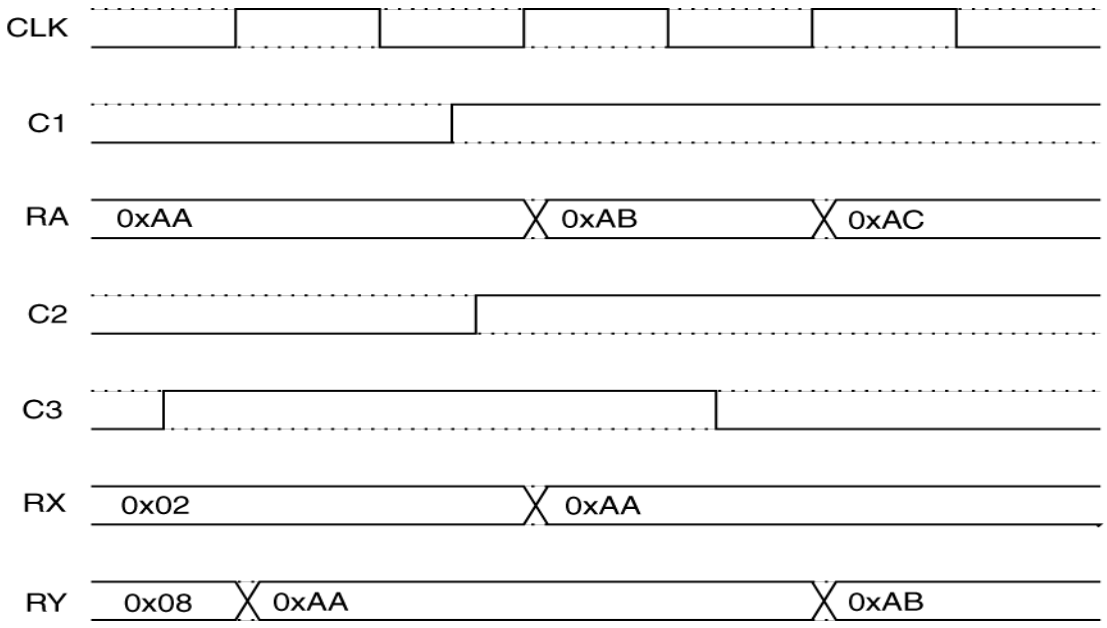


19) Design a circuit that can implement each of the following two RTL statements. Use busses where necessary. All registers are synchronous and contain a control input (LOADx). Registers may also contain other control inputs where necessary. ENx signals are used to control tri-state outputs. Use and any other hardware you deem necessary. The "+" symbol represents an arithmetic addition.

$$\text{ENA, LOADD, LOADC}: \text{RD} \leftarrow (\text{RA} + \text{RC}),\ \text{RC} \leftarrow \text{RD}$$

$$\text{ENB, LOADD, LOADB}: \text{RD} \leftarrow (\text{RB} + \text{RC}),\ \text{RB} \leftarrow \text{RD}$$

20) Design a circuit that could implement the timing diagram shown below. Each of the "registers" in the design is 8-bits wide. You are not responsible for setting the listed initial values. Use any hardware you want. There are three clock cycles shown; provide an RTL statement describing the microoperations that occur at each clock edge.



7 Introduction to Structured Memory

7.1 Introduction

The notion of memory in digital design is a subject that we have been working with in the previous chapters. The previous chapters discussed memory on a relatively small scale such as flip-flops and registers. While those types of memory are important, you typically find other types of memory in digital systems. I like to describe flip-flops and registers as "incidental" memory: I see this as memory that is specific to your relatively small digital system. This chapter introduces the notion of "structured[1]" memory; this type of memory is typically standalone and has significantly more capacity than incidental memory.

As you'll soon find out, when you deal with structured memory, you find yourself learning a new set of approaches and acronyms. The main purpose of this chapter is to introduce and describe the main notions regarding structured memory; a following chapter fills in the details.

Main Chapter Topics

- **OPERATIONAL OVERVIEW OF MEMORY:** This chapter provides an overview of the basic operational and performance characteristics of memory as well as common terminology associated with memory.
- **MEMORY TYPES:** This chapter introduces the two accepted main types of memory, RAM and ROM, by describing their differences and similarities.
- **MEMORY INTERFACE METRICS:** This chapter describes the basic interface issues involved in structured memory device.
- **LOW-LEVEL MEMORY MODELING:** This chapter introduces structured memory by describing a relatively small structured memory device in terms of familiar incidental memory devices.

Why This Chapter is Important

This chapter is important because it provides a basic overview of digital memory and the operation of memory in a digital system.

7.2 Memory Introduction and Overview

This chapter discusses some of the more basic aspects of memory. Once again, this chapter is not an exhaustive approach to the subject, but it hopefully presents a solid overview of the subject. You should refer to other sources for the full story. The unstated goal of these notes is to grasp the basic concepts and basic lingo regarding structured memory. The reality is that there are so many different types of memory out there; a few pages can't possibly go into a significant amount of detail. The hope is that if you're ever faced with

[1] I've adopted this term from the notion of "regular structures", which roughly refers to larger semiconductor devices that have a large and repeated structure that is dedicated to a single purpose. In this case, the purpose is memory.

working with a new memory device, you'll be successful in your endeavor because you'll be roughly familiar with the terms you come across.

Before we start, we need to make one clarification. Often time when we discuss the notion of memory, we sometime use the terms "data" and "information" interchangeably. In most cases, this is no big deal, but you need to understand there is a distinct difference. In the context of digital design, data is nothing more than a bunch of 1's and 0's. Information, however, relates to the *interpretation* of that data. We often refer to data as having information content and there is actually a unit used to measure the information content of data[2]. Roughly speaking, it is up to the user to interpret data as having certain information content or not.

For example, a given set of 1's and 0's may appear random but it could also represent a set of instructions for a given computer. Better yet, a memory unit stores data; if this data represents instructions to a computer, then you could consider the data to be information. On the other hand, if you have a memory that you or anyone else has never written too, the memory is still full of 1's and 0's, but the data has no meaning. The point here is that you need to be on the lookout for misuses of the terms "data" and "information"; it usually does not matter but sometimes it does.

7.3 Basic Memory Operations: READ and WRITE

The two operations that are most often associated with memory are *reading* and *writing.* The notion of a "memory read" or "reading from a memory" refers to the action of retrieving data currently stored in memory; the notion of a "memory write" or "writing to a memory" refers to the action of placing new data in memory. Another way to view reading and writing memory is the copying of data out of memory (reading) and the transfer of data into memory (writing). Reading memory generally does not alter the contents of the memory location you're reading. Writing, on the other hand, does change the contents of memory (namely, the memory location you're writing to).

In an effort to simplify memory operation, memories generally read or write fixed amounts of data from a fixed part of memory. This means that if you need to read various bits stored in different areas of a memory, you're probably going to have to perform multiple reads and further processing in order to assemble the data you're interested in. Memories generally store fixed "chunks" of memory and access to that memory is limited to only that chunk size. If your system requires something different, you'll need to work around you're memory's limitation or change to a different type of memory.

7.4 Basic Memory Types ROM and RAM

There are many different flavors of memory in digital-land; each of these memory types has their own acronym describing them. Despite this relatively high number of memory types, we can classify all of them as either RAM or ROM. The acronyms stand for *random access memory* and *read only memory*, respectively. These terms are rather misleading, particularly in regards to the attributes of modern memory they are describing. In an either effort to classify memories as RAM or ROM, these two acronyms have rather loose but simple definitions. Here is the information embedded in those acronyms.

- The notion of a "read only" memory, or ROM, implies that you'll only be reading from a memory, and never writing to it. Because of this, you'll know that something somewhere at some time wrote to the memory (meaning something filled it with meaningful data). Because the memory is a "read only" memory, you can only retrieve data from that memory; you cannot "easily"[3] alter the data in that memory.

[2] Somewhat unfortunately, the term *bit* is most often used to measure the information content of data. This metric is a function of probability and is not related to the "binary digit" definition of bit that you're more used to using.
[3] Meaning that many types of ROM can be written to; we'll not discuss those cases.

- The notion of a ROM brings up the issue of who put the data that you're interested in into the memory. This is a big issue and it starts delving down into the various sub-types of ROM. We don't want to go there in this discussion because we want to keep this discussion general. The truth is that writing to a ROM is a "special" operation performed by "something". For us, all we're interested in is that there is data in the ROM that we need to read; we'll not worry about writing to it in this discussion.

- The term *random access* is somewhat misleading. This term actually refers to the fact that it takes the same amount of time to access (either reading or writing) each "chunk" of memory stored in the device. While this notion seems rather simple, not all memory devices fall into the category of "random access". The two most obvious notions of non-random access memories are "hard drives" and "tape drives". In particular, the time required to access data in your hard drive is different depending on the physical location of the data on the disk and the current location of the read/write heads. Recall that the hard drive is a mechanical storage device that requires motors and things to move a physical device (the read/write head) radially back and forth across the spinning media to locate the storage locations in question. If the heads are close to the data, it will require less time to access the data than if the head has to first move to the particular location on the spinning disk. The same issue exists with tape drives[4].

- Although the term ROM refers to read only memory, ROMs typically carry the random access attribute. In other words, you can access any of the chunks of data stored on a ROM in an equal amount of time, so ROMs are random access.

- All memories have the notion of being either "volatile" or "non-volatile". If a particular memory is volatile, the data stored in that memory is lost when you remove power from that circuit. Conversely, the data in non-volatile memory is not lost when you remove power.

Despite all these misleading terms and acronyms, RAM and ROM do have accepted definitions. Table 7.1 lists these accepted differences and similarities.

Memory Type	Random Access	Read/Write	Volatile/Non-Volatile
RAM	yes	read & write	volatile
ROM	yes	read	non-volatile

Table 7.1: Accepted attributes of RAM and ROM.

7.5 Memory Operation Details: Reading and Writing

Now that we know that basic memory operations include reading and writing, we now need to examine some of the lower-level details of these operations. Once again, we keep this discussion general in that every memory out there has its own version of reading and writing; you'll need the access the datasheets associated with those memories for the full story.

Figure 7-1 shows a high-level diagram of a memory device. We'll keep this discussion centered on integrated circuit semiconductor memories as items such as hard drives and tapes drives are harder describe in general terms. Figure 7-1 shows that we can classify the various signals associated with interfacing with a memory

[4] With the onset of large and inexpensive semiconductor memories, tape drives are less popular, particularly since semiconductor memory has become more popular. We'll not waste time describing it here.

device into three categories: address lines, data lines, and control lines[5]. The following is a general overview of these lines. In general, the widths of these bundles are associated with the specific capacity attributes of the memory; we'll deal with those issues soon.

Data Lines: The data lines are a set of signals that route the bits you're writing or reading into or out of the memory device. Note that the arrow associated with the data lines has an arrowhead on each end, which typically signifies that data on those particular lines can travel either into the memory (for write operations) or out of the memory (for read operations)[6]. The data lines are either serial or parallel; the slash notion in Figure 7-1 implies that we are dealing with parallel data lines. The parallel lines handle the transfer of a particular chunk of memory simultaneously, which is where we'll focus our discussion. Figure 7-1 happens to show only one set of data lines; in reality, memories often have both a set of input data lines and output data lines.

Address Lines: The address lines are a set of signals that provide the memory with a "location" within the memory to write to (write) or read from (read). Keep in mind that we're describing random access memory, which means all the various chunks of data stored in the memory are accessible in the same amount of time. The address lines are the method that the memory uses to differentiate one chunk of memory from another on the interior of the device.

Control Lines: The control lines are a set of signals that determine and direct the various operations associated with the memory. The best example of the responsibility of the control lines are with RAM devices that are both readable and writeable; the control lines allow the user to control which operation occurs. The underlying notion of control lines is that simple memories have few control lines; memories that are more complex have more control lines[7].

We'll soon delve further into the details of memory interfacing. For now, you can consider the general interfacing operation of a memory read as: 1) give the memory an address, 2) tweak the control lines, and 3) wait for the data. For memory writes, you generally 1) give the memory an address, 2) give the memory the data, and 3) tweak the control lines. It's that simple. Or is it?

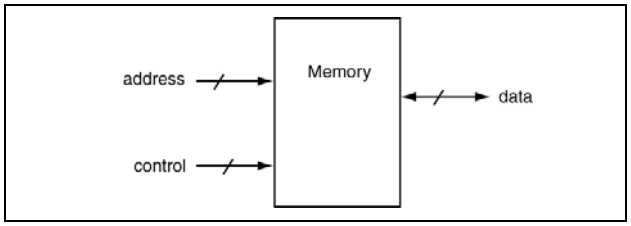


Figure 7-1: A general diagram of a memory integrated circuit.

7.6 Memory Specification and Capacity

When working with memory and memory systems, the two most important pieces of information are the capacity of the memory and the speed of the memory. The memory capacity refers to how many bits the memory can store while the memory speed refers to how fast we can tweak (read and/or write) that memory.

[5] In this context, the notion of "lines" refers to a bundle of wires or signals. You often hear the term "lines" associated with standard bundles such as "data", "address", and "control" lines.
[6] It is not possible for data to simultaneously in both directions.
[7] In an effort to increase memory capacity while keeping physical size small, interfacing some modern memories have become rather complicated and thus have a relatively large number of control signals.

People in digital-land describe memory capacity in many different ways; most of these ways are rather confusing. As is typically in any human oriented pursuits, people attempt to make their "thing" sound better than reality. This applies to memories with the notion of trying to make the memory sound bigger than it really is. While these statements are not lies, they are misleading. You, the digital designer must be able to see through the smoke and hand waving and understand the characteristics of the memory you're working with.

While we know that memory stores bits, and these bits are stored at certain addresses within the memory, memories are rarely bit-addressable[8]. In other words, specific memory devices only allow you to address larger chunks of data. The implication here is that if you need to read or write a single bit, your only choice is to work with the minimum chunk of addressable data as specified by that memory device. The fact that a memory is not bit-addressable is generally not a bad attribute; there are always ways to work around the fact that you may need only a single bit. Additionally, making memory bit-addressable would create a horrendously inefficient device. Because of these issues, memories generally compromise by providing data only in some reasonable sized chunk[9].

Data in memory is most often stored in groups of bits; the most common term we typically use to describe these ground of bits is a *word*. The official definition of a word is the smallest addressable unit (or chunk of bits) in a memory or memory system[10]. This term is important because we typically described memories and memory systems in terms of words rather than bits. Referring to memory in terms of the smallest addressable chunk of memory is the honest and most understandable approach; too bad more people aren't honest[11].

Figure 7-2 shows a diagram of a generic memory including some typical memory characteristics. This is an important diagram as the metrics in this diagram are typical of most memory devices. Here is an overview of the most important aspects of Figure 7-2 while Table 7.2 summarizes all the gory details.

- The by "2^m x S" notation is usually how people state the capacity of a memory. The underlying notion is that we are modeling the memory as a two-dimensional grid, as the "x" in "2^m x S" indicates. The important aspects are what the terms on both sides of the "x" represent.

- Practically everything having to do with memories relates closely to the notion of binary. The term "m" refers directly to the width of the address bus or address lines. This means that the number of memory pieces that a memory can access is the number two raised to some integral power. Datasheets do not directly show the exponential term in Figure 7-2, but you can represent the term they do show with an exponential such as "2^m".

- The term "S" is the width of the data bus or data lines. Datasheets usually state this metric in bits, but sometimes they state it using bytes or some larger form of bit aggregates. Generically speaking, we consider the terms "S" as the word size for a particular memory. The true capacity of a memory (the amount of data it can store) relates directly to the number of address lines.

- The total word storage capacity for the memory is how many words the memory can store. For this particular memory, the word storage capacity is thus "2^m".

- The total bit storage capacity for the memory is a product of the number of words and the number of storage locations in the memory. Thus the bit storage capacity is given by "2^m x S".

[8] This is not exactly true. There are bit-addressable memories out there, but they are typically grouped together to form memories of different capacities, namely different bit-widths. More on this in a later chapter.
[9] Typically, people attempt to make their memories sound bigger by stating how many bits the memory can store even though that particular memory is not bit-addressable.
[10] We keep stating "memory or memory system" because you'll soon find out that you can design "memory systems" of specific capacities using many smaller memory modules. This means you have the ability to transcend the basic limitations of a single memory module by cleverly designing a "memory system". This is a topic of a later section in this chapter.
[11] Note that I have not inserted an "academic administrator" comment here…

- The control lines do not include a bundle width indication. We do this in order to keep the discussion as general as possible. The notion of "2^m x S" is common and specific; the control lines for memory modules tend to vary greatly across different devices.

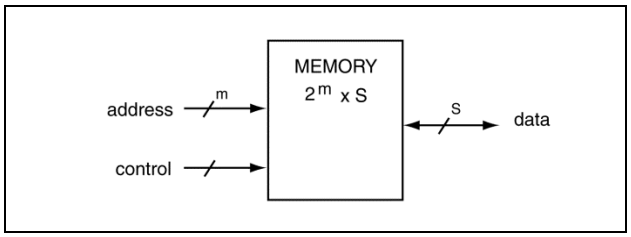


Figure 7-2: A diagram of memory indicating notions of storage capacity.

$$capacity\ (in\ bits) = 2^m \cdot S$$

Equation 7-1: Closed form formula for memory storage capacityin bits.

Symbol	Definition
m	Bit-width of address bus
S	Bit-width of data bus
2^m	Memory capacity in words
2^m x s	Memory capacity in bits

Table 7.2: Summary of memory definitions and properties.

7.7 Memory Interface Metrics

Here's a quick overview of the timing and control issues associated with interfacing a memory. Recall that a memory write transfers a word to be stored in memory while a memory read prompts a memory to output the contents of memory. The actual reading and writing of memory is control by the so-called "control lines" associated with the memory device. Every memory has its own method of reading and writing; specifically, each memory has its own protocol for tweaking the control lines in such a way as to obtain the desired function from the memory device. This section examines the control lines and their relation to the data and addresses lines for basic read and write operations on a generic memory. Read and write operations are roughly similar though write operations are slightly more involved, as we'll see in later timing diagrams.

Memory Writes: For a memory write operation, you will provide the memory with data that will overwrite data currently stored in the memory. The information on the address lines provides the location of where the word will be stored. The bits on the data lines provide the data that will transfer and store on the memory device. The write operation necessarily overwrites the data currently stored at the address indicated by the information on the address lines. If you follow the proper protocol, your write operation will be successful.

Memory Reads: For a memory read operation, you will prompt the memory device to output the data currently stored at a specific location in memory. The information on the address lines provides the location in memory of where you want to read from. Thus, the address lines provide the memory location of the word that will transfer out of the memory; this transfer occurs by placing the data at the specified address onto the data lines. Read operations

generally do not alter values currently stored in the memory device. If you follow the proper protocol, your write operation will be successful.

Steps for Memory Writes	Steps for Memory Reads
1. Apply the information representing the memory location of where you desire to *store* the given word to the *address* lines. 2. Apply the information representing the actual data bits to be *written* to the *data* lines. 3. Tweak the control lines to make the *write* operation occur.	1. Apply the information representing the memory location of where you desire to *retrieve* the given word to the address lines. 2. Tweak the control lines to make the *read* operation occur.

Table 7.3: Summary of generic steps required for memory reads and writes.

7.8 Memory Performance Parameters

The previous discussion did not say much about the control lines. The section provides more information regarding control in the context of typical memory performance parameters. Keep in mind that when we speak about memory devices, we're talking about actual physical electronic devices. This means that read and write operations require finite and specific amounts of time for them to successfully occur. Most of the associated performance parameters are outside the scope of this discussion, but some are basic and popular enough for a mention and overview here.

Figure 7-3 and Figure 7-4 show timing diagrams associated with typical read and write operations, respectively. These two figures agree with basic read and write descriptions from the previous section. While you're familiar with the address and data lines associated with a typical memory, Figure 7-3 and Figure 7-4 show specific control signals what we'll describe below. Note that these signals have somewhat popular acronyms in terms of memory devices, but the choice is still arbitrary. In reality, Figure 7-3 and Figure 7-4 could be representing one memory that is both readable and writeable; this memory device would thus contain three control lines. For this mystical device, the number of address and data lines in immaterial for this discussion.

- Figure 7-3 shows a timing sequence for a memory read operation. Two control signals, CS and OE, control the read operation. The CS signal is a popular digital acronym that stands for "chip select" while the OE is stands for "output enable". As Figure 7-3 indicates, in order to read data from the memory, we need to assert the CS and OE lines. The timing diagram in Figure 7-3 also shows that we have asserted the correct address lines at the same time as asserting the CS and OE control lines. Sometime after the assertions of the CS and OE lines, the correct data becomes available on the memory's data lines.

- Figure 7-4 shows a timing sequence for a memory write operation. In addition to the CS signal, this memory device also has a WE control signal. Roughly speaking, the WE signal officially writes the data on the data lines to the memory location specified on the address lines. In Figure 7-4, we've simultaneously asserted the address lines and the CS control line, which is arbitrary. The data on the data lines becomes ready to write sometime after that, which is also arbitrary. Also implied by Figure 7-4 is the notion that once the data officially writes to the memory, we de-assert the WE line.

Generally speaking, signals such as WE must remain asserted for a specific amount of time in order for the write operation to be successful. Check a datasheet for details.

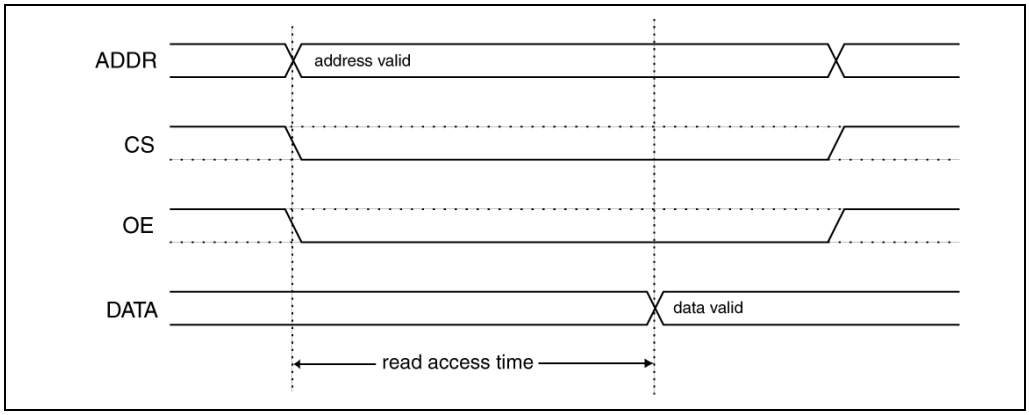


Figure 7-3: A typical control sequence for a memory read operation.

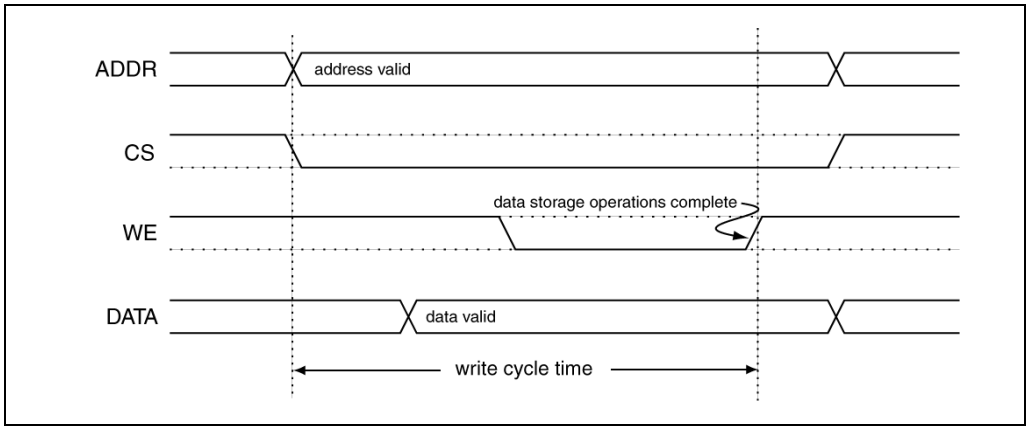


Figure 7-4: A typical control sequence for a memory write operation.

Because a memory is physical device, successful read and write operations require a finite amount of time to occur. Recall that the two main design parameters associated with a memory device are its capacity and its operational speed. We've previously discussed memory capacity issues; we now need to discuss a few of the more common memory performance issues and metrics.

As with just about everything in digital-land, the faster something can operate, the more highly regarded that device it[12]. This is maybe even more so true with structured memory devices as they are typically a major component in many digital systems, particularly computer systems. Moreover, in many digital systems, more than one device in the system must access memory. Often times more than one device must simultaneously access memory; this situation creates what we refer to as a "bottleneck. This condition is undesirable in the one or more devices must wait to access memory[13]. The notion of "waiting" in digital-land means your device is probably doing nothing, thus probably lowering the overall throughput of your system. Roughly speaking,

[12] It's typically more expensive also.

[13] There is a notion of "multi-port" memories. These memories typically allow some type of parallel operation such that two devices can simultaneously read from two different memory locations. These types of memories become expensive and certainly exercise the inherent trade-offs in digital systems designs.

the faster your memory operates, the less chance you have of a bottleneck; or if you have a bottleneck, it won't be a severe as if you had a slower memory.

There are three main parameters we use to describe memory performance. These parameters state how fast you can read from memory (read access time), how fast you can write to memory (write cycle time), and roughly how much data you can pass back and forth to and from the memory (bandwidth). Figure 7-3 and Figure 7-4 show graphic examples of the read access and write cycle times, respectively. The list below provides a more detailed description of these three performance parameters. Note in the descriptions below that we do not include details regarding the control signals in order to keep the description general.

- Memory Read Access Time: The minimum time required to access a word from memory. Specifically, this is the amount of time measured from the application of a valid address to the address lines to the appearance of the associated data on the data lines.

- Memory Write Cycle Time: The minimum time required to write a word to memory. Specifically, this is the time measured from the application of a valid address to the address lines to the completion of the internal operations required to successfully store the data in memory.

- Memory Bandwidth: The maximum data transfer rate in a given amount of time data associated with a memory. Since both read and write operations require finite amounts of time, it's worthwhile knowing the amount of data that we can physically transfer to and from memory in a given amount of time.

Any time you need to work with a new memory device, you'll find yourself concerned with the above parameters. There actually much more to it than this but this is only an introduction. Probably one of the most informative items regarding working with memory devices is the associated timing diagram, which you can generally find in the associated datasheet. There is almost a special language used to specify all the timing parameter associated with memory devices. Once you start reading about and working with actual memories, you'll quickly get the hang of things.

7.9 Low-Level Modeling of a Simple Memory

It's not overly complicated to model a structured memory using devices that you are already familiar with. As you can probably imagine, we can model a structured memory device in all its glory using basic sequential and combinatorial devices from your digital bag of tricks. Let's go through the steps of generating the simple memory shown in Figure 7-5. Of course, this is not an overly useful device due to its relatively small capacity, but it will illustrate the more important points in the world of structured computer memory.

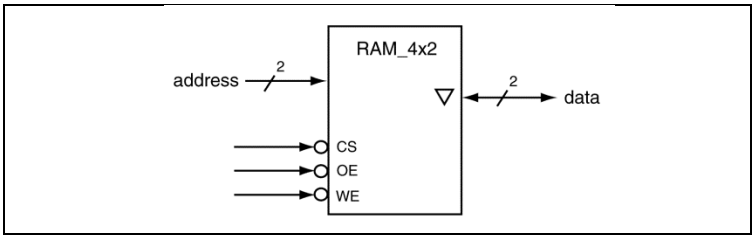


Figure 7-5: Block diagram of a 4x2 memory device.

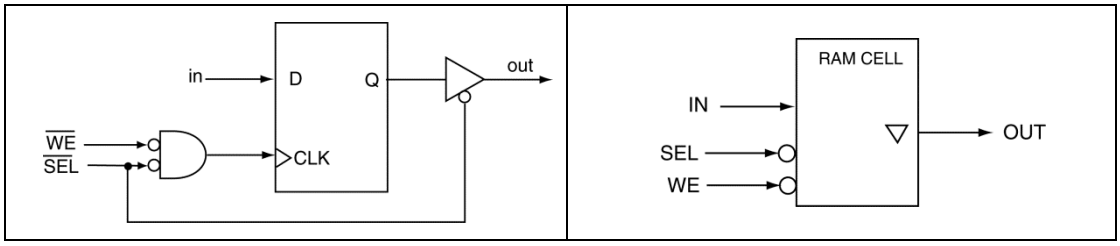
Figure 7-6(a) shows the basic single-bit memory cell we'll for our structured memory design. This single-bit cell contains both sequential and combinatorial circuit elements with the D flip-flop and supporting control logic, respectively. The D flip-flop is the memory element for the cell. The supporting control logic for the cell

includes an active-low tri-state buffer on the output and a NOR gate[14] on the input. There are two active-low control signals for the cell. WE roughly stands for "write enable" and partially controls the ability of the device to latch data into the flip-flop while the SEL signal controls the ability to write to the cell and also controls the tri-state enable on the cell's output buffer. Here are more details regarding this memory cell's operation.

- The state of the SEL signal must be a '0' in order for the storage cell's contents to appear on the output because the enable control of the output buffer is active low. The SEL signal must be also be asserted ('0' state) in order to write data to the D flip-flop. In other words, if SEL is a '1', the output of the memory cell is in a high-impedance state.

- Writing to the device requires a rising clock edge; the control logic synthesizes this output when both of the WE and SEL signal are simultaneously asserted[15]. The D flip-flop requires a rising edge; you can create this edge by simultaneously asserting WE and SEL or asserting one of these signals and them the other. Moreover, a read of the device's contents needs to have SEL asserted and WE not asserted while a write needs to have both SEL and WE asserted. This particular control arrangement is arbitrary. The information in Table 7.4 provides an outline of how you should operate this memory cell. In reality, these particular controls allow you to both simultaneously read from and write to the memory cell.

The best approach is to working with the memory cell in Figure 7-6(a)is to keep read and write operations separate, which is how Table 7.4 lists them. Figure 7-6(b) shows a block diagram of this device, which we abstracted to a higher level.



(a) (b)

Figure 7-6: The basic memory cell (a) and its block box representation (b)

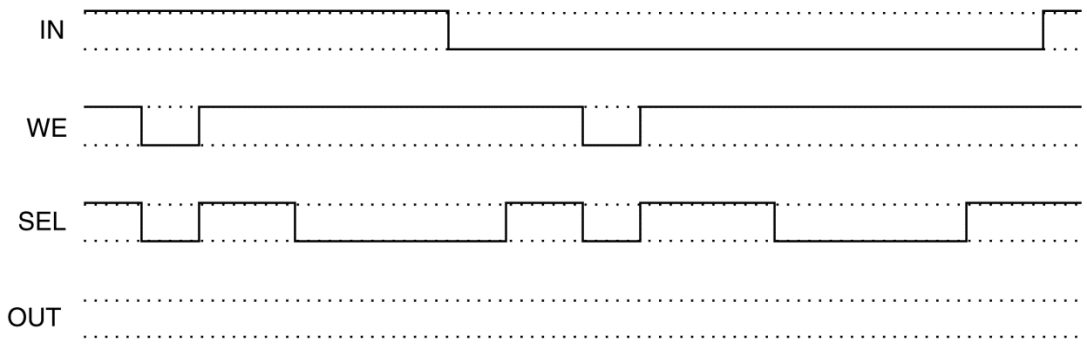
Signal	Memory Cell Reads	Memory Cell Writes
SEL	Assert the SEL signal. *This enables the tri-state device.*	Assert the SEL signal. *This enables the input NOR gate.*
WE	Un-assert the WE signal. *This prevents writing to flip-flop by disabling clock signal.*	Assert the WE signal. *This enables the input NOR gate.*

Table 7.4: Summary and explanation of actions required for cell reads and writes.

[14] Note that this is the AND form of a NOR gate.
[15] In this case, you can consider the SEL in its '0' state "asserted" in context of the input control logic.

Example 7-1: Memory Cell Timing Diagram

Complete the following timing diagram using the memory cell diagram shown in Figure 7-6(a). For this problem, assume there are no circuit delays associated with the circuit.



Solution: Figure 7-1 shows the final solution to this example. Here are a couple of useful things to note about this solution.

- The first part of this solution is to realize that the problem states the initial condition of the memory cell's flip-flop to be '0'. Although you would think this would be important, it is unimportant for this problem because of the fact that the output buffer is disabled until a new value is written to the cell.
- The cell's output is in a high-impedance state when the SEL signal is not asserted. The timing diagram chooses to represent high-impedance with a signal that is neither high or low. This is one of many ways to model high impedance with the thought being that we can't be sure of the signal's value so we place in the middle of the '1' and '0' outputs.
- Writes to the cell requires the assertion of both WE and SEL. Reads from the memory cell require the assertion of only the SEL signal.

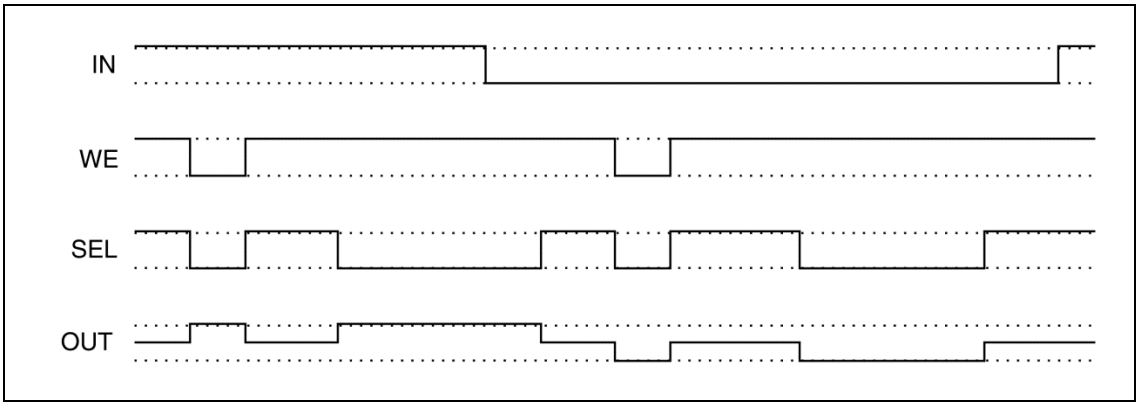


Figure 7-7: The solution to Example 7-1.

The next step in the design is to provide the all the required memory cells in order to store the amount of bits required for the 4x2 RAM. The final RAM's capacity is eight bits, so we need to start our design with eight

memory cells. The word size of this particular memory is two so we'll need to configure these devices in such a way as to make only two of the RAM cells active at one time. The accepted approach to doing this in digital-land is to use a standard decoder. The standard decoder has the desired output characteristic required by the array of RAM cells because only one of the decoder's outputs is asserted at any one time. We connect each of the decoder's output to a "row" of RAM cells. Note that there are two cells per "row" and the two cells in any given row form the "word" output of the final RAM. Recall that we typically refer to memories in by their basic row/column structure. The basic characteristics of a standard decoder support this row/column view of the RAM.

Figure 7-8 shows the configuration of the RAM cells including the standard decoder. The decoder effectively only enables one row of the RAM cell array at one time. We configure this RAM such that the rows represent the individual words store in memory. The columns in Figure 7-8 represent the aggregate of same bit locations of the various words stored in the RAM. Note that his RAM is thus only row-addressable, which implies that the RAM is not bit-addressable.

The decoder has two inputs, which allow the devices to enable one of the four rows in the RAM. The standard decoder's two inputs thus become the official address of the rows within the RAM, and thus the address of the data in the RAM itself. For example, if you need to address the second row from the top, the decoder's inputs would be ADDR(0) = '1' and ADDR(1) = '0'. In relation to Figure 7-8, the upper-most bit-storage cells form the lowest-order word memory (ADDR = "00") while the lower-most bit-storage cells form the highest-order word in memory (ADDR = "11").

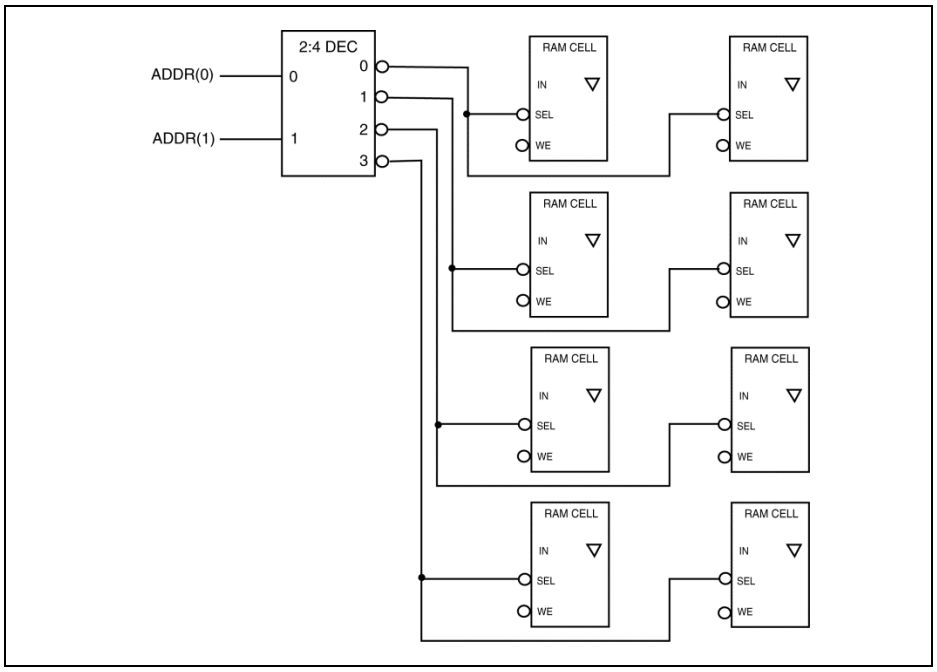


Figure 7-8: Basic memory cell configuration with associated decoder.

The next step in the design process is to configure the outputs. Note that the outputs are bi-directional at the level we'll currently modeling them at so we'll need to deal with a tri-state buffer in the configuration. The first part of this step is to connect the columnar outputs of the RAM cells to a single tri-state buffer; Figure 7-9 shows this result. We'll use a single tri-state buffer for each column; the outputs of these buffers become the data outputs of the 4x2 RAM. Remember that only one row of the RAM configuration can be active at one time so connecting the RAM cell columns to a single output does not cause contention on the line and is thus not problematic. In official terms, there would never be a case where there was contention on the output lines; using a standard decoder ensures that contention does not occur.

To complete this step we will need to configure the data inputs to the RAM. We achieve the desired bi-directionality of the RAM by tapping into the output of the RAM's output buffer and use the signal as input. This type of connection is typical for devices that have tri-state outputs. This works because you'll never be simultaneously reading data and writing data to the RAM. Figure 7-9 shows the final configuration after this step. Each word is stored at the two-bit address associated with the ADDR lines while D lines (two bits) are the data output of the RAM.

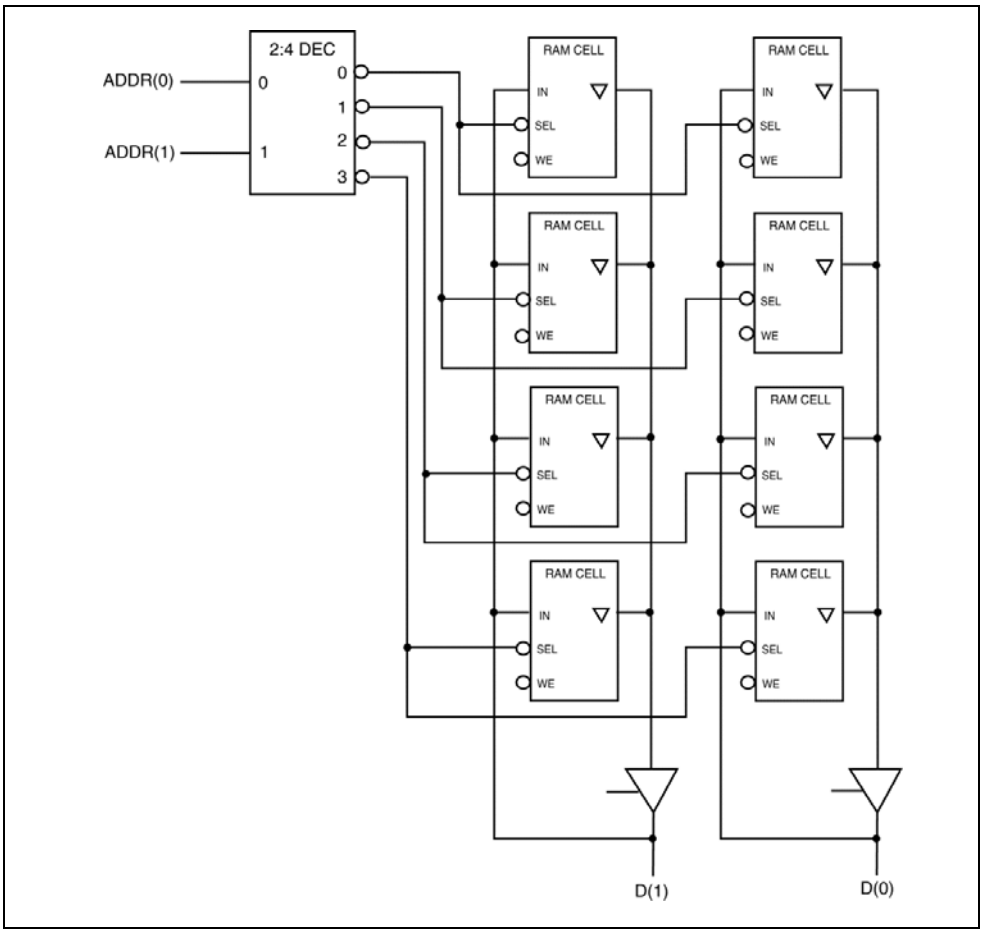


Figure 7-9: Adding in the input and output to the basic memory configuration.

The final step is to add in the control signals. Keep in mind that purpose of these signals is to control the reading and writing of words to memory. The read signal needs to drive the desired word out of the RAM while writing places new words of data into the RAM. The first troubling thing you may notice about the circuit diagram in Figure 7-9 is that the cell inputs appear to connect to the cell outputs. This style of connection will not be a problem because we'll add in the controls to the tri-state devices in this step.

There are three control signals for this RAM device. The following blurbage describes their basic functionality. This description works in conjunction with the final circuit diagram in Figure 7-10.

> **CS:** The CS signal acts as an active low "chip select" for the RAM. When the CS is not asserted, the two AND-looking gates on the circuit's inputs are both dead. The implication of this configuration is that the CS signal needs to be asserted in order for you to read from or write to the RAM. In other words, if you want to use this RAM, you need to assert CS.

OE: The OE signal is an active low "output enable": for the RAM. The OE signals works in conjunction with the CS signal to allow read operations from the RAM. Both the CS and OE signals need to be asserted in order to allow the RAM's output buffers to drive a word of memory (a row in the RAM) onto the RAMs data output lines.

WE: The WE signal is an active low "write enable" for the RAM. The WE signal works in conjunction with the CS signal to allow write operations to the RAM. Both the CS and WE signal need to be asserted in order to allow data to be written to the RAM's underlying memory elements.

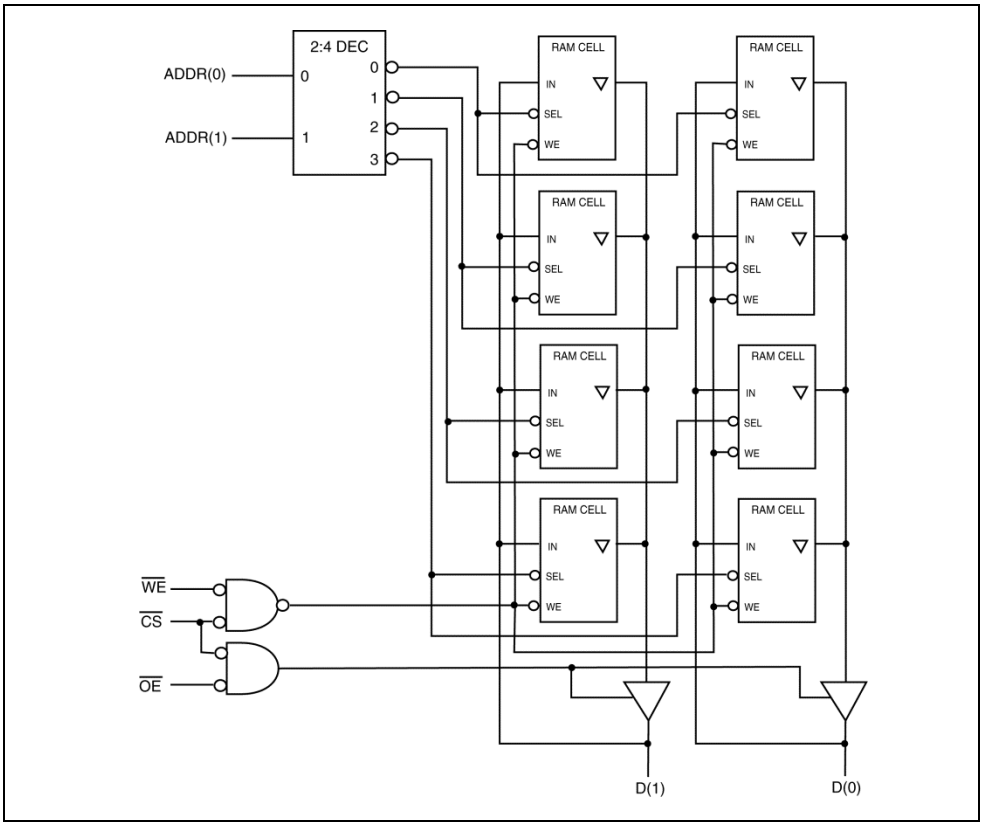


Figure 7-10: The final memory with addition of controlling logic.

Once again note that both the WE and OE signals are only effective when they are asserted in simultaneously with the CS signal. It's also good to note that you must use the control signals with caution, as it is wholly possible to assert the control signals such that you're attempting to both read and write from a particular address. The problem with this notion lies in the RAM's basic circuit design. The OE enables the output buffer's tri-state control, which drives memory contents on the output lines. If some other external device is also driving the output lines, there will be contention on the lines. In this case, you'll not know what data actually writes to the RAM. The final word on this circuit is that it is up to the digital designer to ensure that both the OE and WE signals are not simultaneously asserted.

7.10 Chapter Summary

- Memory is a form of a sequential circuit, but we further divide memory into two categories: “incidental memory” and “structured memory”. Incidental memory refers to items such as flip-flops and registers (relatively small) while structured memory refers to larger capacity regular structures.

- There are many type of memory in digital-land, but we can roughly classify them all as either ROM or RAM. ROM is “read only” memory while RAM is “random access” memory. Both of these memories have the random access attribute in that all of the data on the devices is accessible in the same amount of time. ROMs are considered non-volatile while RAMs are not. RAMs can be both written to and read from while ROM can only be generally read from.

- The notion of reading from a memory, or a memory READ, consists of making the data within the memory at a given address available to entities external to the memory. Memory reads generally do not alter the data stored in the memory. The notion of writing to a memory, or a memory WRITE, consists of overwriting data contained in the memory at a given address with data provided by some entity external to the memory.

- Interfacing with memory generally requires tweaking one the three types of I/O associated with memory. The three types of memory I/O are address lines, data lines, and control lines. The address lines provide an index into the memory and allow access to a particular chunk of data stored in memory. The data lines provide a path for data to flow into (write) or out of (read) memory. The control lines provide a structured approach to read from and/or writing to the memory device.

- Memories are generally rated by the capacity (how many bits they can store) and the speed (how fast you can read and/or write the memory). The term “word” is used to refer to the smallest chunk of memory available at a given address in the memory. Memory capacity can be stated in bits or words; any other approach is suspect as it can be misleading

- Memories typically store two raised to an integral power number of words. The integral power in this case is the number of address lines on the memory. The number of address lines is sometimes referred to as the width of the address bus.

- Memory speed is rated by how fast you can read from it and/or write to it. The term “read access time” refers to how fast you can read from a memory. The term “write cycle timing” refers to how fast you can write data to a memory. The term “memory bandwidth” refers to the maximum amount of data going to and coming from a particular memory in a given amount of time.

7.11 Chapter Exercises

1) In your own words, describe what is meant by the term "random access" in the context of computer memories.

2) In your own words, describe what is meant by the term "volatile" in the context of computer memories.

3) In your own words, describe the accepted functional differences between RAM and ROM.

4) Complete the following table for the given memory specifications. Assume each of these memories is contained on a single integrated circuit.

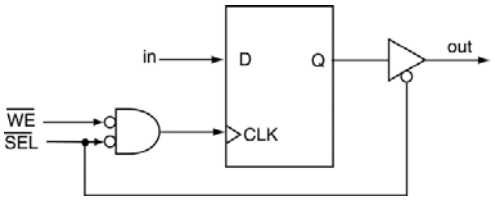
Memory Specification	Address Bus Width	Memory Capacity in Bits	Memory Capacity in Bytes
256 x 8			
256 x 24			
1K x 16			
2K x 8			
8K x 32			
32K x 12			
64K x 16			
256K x 8			
1M x 16			
4M x 32			
8M x 8			
64M x 48			
128M x 32			

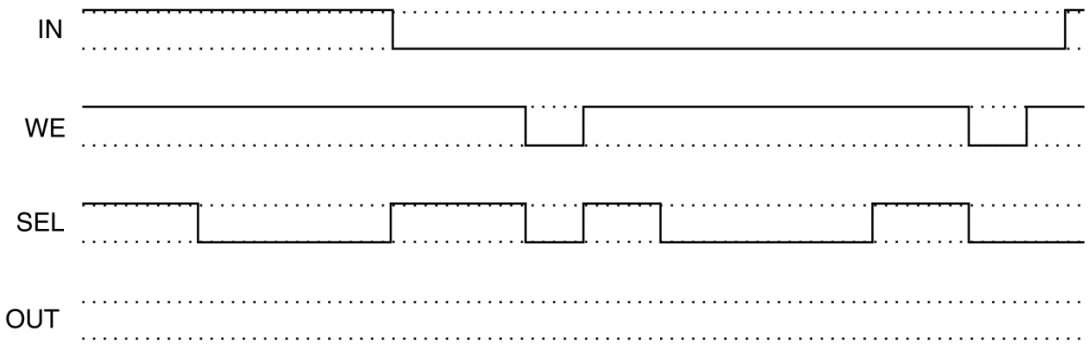
5) In your own words, explain how read and write access times affect the bandwidth of a given memory.

6) Describe a circuit situation where having a large memory bandwidth would be important.

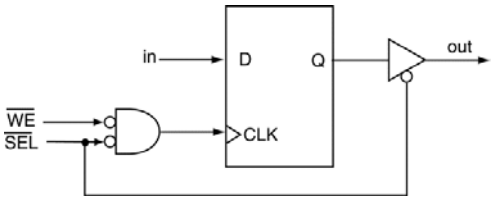
7) Faster memories are typically more expensive than slower memories. Speculate on why you feel this would be the case.

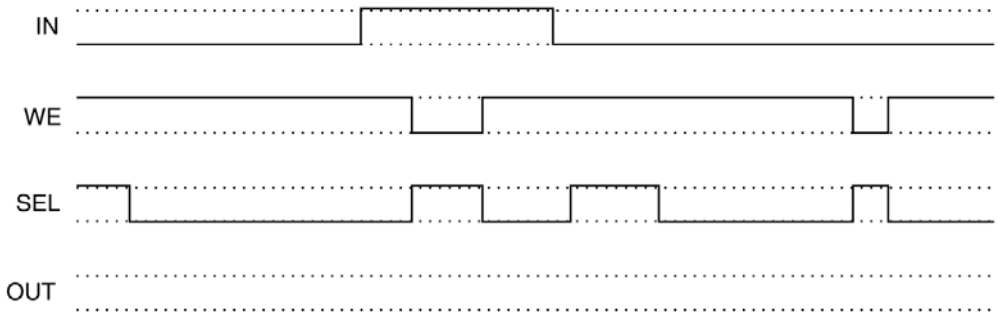
8) Complete the following timing diagram using the memory cell diagram shown in on the right. For this problem, assume there are no circuit delays associated with the circuit. Assume the memory cell is initially storing a '1'.





9) Complete the following timing diagram using the memory cell diagram shown in on the right. For this problem, assume there are no circuit delays associated with the circuit. Assume the memory cell is initially storing a '1'.





10) A given RAM capacity is specified as 1Kx24.

- List the capacity of this RAM in both bits and bytes.
- List the number of address lines this RAM would contain.

11) A given RAM capacity is specified as 1Kx32.

- List the capacity of this RAM in both bits and bytes.
- List the number of address lines this RAM would contain.

12) A given RAM capacity is specified as 8Kx32.

- List the capacity of this RAM in both bits and bytes.
- List the number of address lines this RAM would contain.

13) A given RAM capacity is specified as 16Kx24.

- List the capacity of this RAM in both bits and bytes.
- List the number of address lines this RAM would contain.

8 Structured Memory

8.1 Introduction

The previous chapter introduced many of the important ideas regarding memory in digital circuits. Recall that we introduced the concepts of "incidental" memory and "structured[1]" memory. This chapter delves further into the concepts of structured memory and examines several approaches to dealing with this type of memory. If you continue as a digital designer, it's inevitable that you'll need to work with structured memory. The basic principles don't change, but the low-level details of interfacing with the memory you need to work will require you to spend some time with your head buried in the datasheet.

The notion of structured memory in digital-land used to be rather simple. Way back in the early years of digital design, there was not that many options available for structured memory. In addition, the memory options that were available were relatively simple to understand and use. The drive to make memory faster and larger combined with the notion of creating "special memory" for specific uses has made modern memory into a vast subject that can become extremely complicated. This chapter makes no attempt to describe all the memory out there; this chapter describes only the most basic issues involved with structured memory including basic memory systems design and memory modeling using VHDL.

Main Chapter Topics

- **STRUCTURED MEMORY MAPPING & MEMORY SYSTEMS DESIGN:** Memory systems are typically designed as many individual memory units as opposed to one single unit. This form of design requires a unique high-level perspective of the system and uses standard digital devices in their implementations. This chapter provides an overview and introduction to structured memory system design.
- **VHDL MEMORY MODELING:** It is possible to model structured memory using many different approaches in VHDL. This chapter outlines VHDL models for a ROM, RAM, bi-directional RAM, and a dual-port RAM.

Why This Chapter is Important

This chapter is important because it provides a basic overview of structured memory and various details of modeling and interfacing with that memory.

8.2 Memory Mapping

You can typically find the notion of memory mapping in microcontroller (MCU) and microprocessor-based (MPU) systems. The idea is that different areas of memory are typically used for different purposes in most digital systems. The notion of a MCU or MPU-based system generally implies that the system is becoming

[1] I've adopted this term from the notion of "regular structures", which roughly refers to larger semiconductor devices that have a large and repeated structure that is dedicated to a single purpose. In this case, the purpose is memory.

relatively complex. The segmenting of memory, or the designation of certain parts of memory to specific purposes, aids in the overall understanding of the system. This makes it well work looking at in this chapter.

The notion of memory mapping is an exercise in the study of binary and hexadecimal numbers. Even a simple digital system is large enough not to deal with binary values and ranges, so we quickly convert to hexadecimal notation to describe memory mapping. The use of hex notation is not complicated, but it is somewhat of a language all its own. The good news is that once you see a few tricks and work with it for a while, you'll for sure find it rather straightforward.

The main idea behind memory mapping is to use multiple smaller chunks of memory space to create a larger memory space. The larger memory space is not "full" in all cases, as there may be areas of memory space that have no physical memory. Under these conditions, the thing we're most interested in is the address ranges associated with a particular chunk of memory as it relates to the larger chunk of memory. The best approach to understanding these issues are with a few example problems. It grinds down to a binary math problem, which heavily implies tweaking around with powers of two.

Before we proceed, let's show some important numbers regarding binary number ranges and their hexadecimal representations. Table 8.1 shows the relation between the number of address bits of a given memory and the associated address range. The first column in Table 8.1 shows the number of address bits associated with a given memory while the other three columns show the zero-based address ranges possible from those given address bits. Note that the decimal representations quickly become barely perceptible. We don't even bother writing out the binary equivalents, as we would quickly inundate your brain with 1's and 0's.

There are a few other important things to realize about Table 8.1. The "Address Range" column provides the associated address range in an 8-digit hexadecimal format. Note the maximum address in any range is associated with all the address bits being at a '1' value. This subsequently provides the "1→3→7→F" format associated with the first non-zero digit reading from left to right. Also note for both the third and fourth columns of Table 8.1 that the number ranges double as you proceed downwards in the table. This is a by-product of the underlying binary nature of memories.

# of Address Bits	Decimal Range	Address Range (hexadecimal)	Abbreviated Range
1	0-1	0-00000001	-
2	0-3	0-00000003	-
3	0-7	0-00000007	-
4	0-15	0-0000000F	-
5	0-31	0-0000001F	-
6	0-63	0-0000003F	-
7	0-127	0-0000007F	-
8	0-255	0-000000FF	-
9	0-511	0-000001FF	-
10	0-1023	0-000003FF	0-1K
11	0-2047	0-000007FF	0-2K
12	0-4095	0-00000FFF	0-4K
13	0-8191	0-00001FFF	0-8K
14	0-16383	0-00003FFF	0-16K
15	0-32767	0-00007FFF	0-32K
16	0-65535	0-0000FFFF	0-64K
17	0-131071	0-0001FFFF	0-128K
18	0-262143	0-0003FFFF	0-256K
19	0-524287	0-0007FFFF	0-512K
20	0-1048575	0-000FFFFF	0-1M
21	0-2097151	0-001FFFFF	0-2M
22	0-4194301	0-003FFFFF	0-4M
23	0-8388607	0-007FFFFF	0-8M
24	0-16777215	0-00FFFFFF	0-16M
25	0-33554431	0-01FFFFFF	0-32M
26	0-67108863	0-03FFFFFF	0-64M
27	0-134217727	0-07FFFFFF	0-128M
28	0-268435455	0-0FFFFFFF	0-256M
29	0-536870911	0-1FFFFFFF	0-512M
30	0-1073741823	0-3FFFFFFF	0-1G
31	0-2147483647	0-7FFFFFFF	0-2G
32	0-4294967295	0-FFFFFFFF	0-4G

Table 8.1: Number of bits and associated number ranges.

Example 8-1: Memory Mapping with Two Devices

Show the address ranges in both binary and hexadecimal associated with the use of two 64x8 memories to form one 128x8 memory.

Solution: The first part of this solution is to draw a diagram to enhance our understanding of the sparsely worded problem. Figure 8-1 shows a diagram we'll use to solve this problem. As you can see from Figure 8-1, we've virtually attached two 64x8 memories, which as the effect of forming a 128x8 chunk of memory[2].

[2] We could have created a 64x16 memory, but that is a topic we'll deal with in a later section.

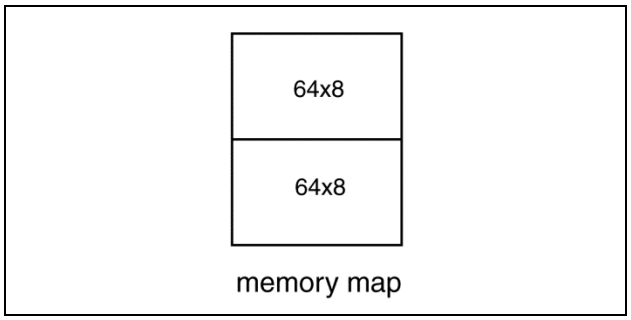


Figure 8-1: The diagram associated with Example 8-1.

The first thing to note is that a 64x8 memory uses 6 bits for its address lines. Indexing 6 bits into Table 8.1 shows that associated range is 0→3F, or "000000"→"111111". The using two of these memories would create twice the capacity of a 64x8 memory, for an overall memory capacity of 128x8. In other words, you can model two 64x8 memories as one 128x8 memory. Indexing into Table 8.1, you can see that a 128x8 memory is associated with seven address bits. The only difference between these two memories in the 128x8 configuration is the most significant bit of the virtual 7-bit address; the six lower bit ranges will are essentially equivalent. The final address value for the lower-order memory in Figure 8-1 is thus '0' appended to the 64x8 range while the final address value for the higher-order memory is '1' appended to the 64x8 range. In other words, pasting two memories together requires some other mechanism for differentiating between the two memories, which we do quite easily by adding one more bit in the most significant bit position.

Figure 8-2 shows the final solution to Example 8-1. The hexadecimal numbers on the left side of the diagram show the range of associated address values; the numbers on the right side show the binary equivalent to the hexadecimal values. This is a massively important diagram for several reasons. First, note how the 1-bit values change on the memory boundaries: the MSB becomes a '1' and the other bits all become '0'. Secondly, notice that the original 6-bit ranges are the same for both the lower and higher-order memories.

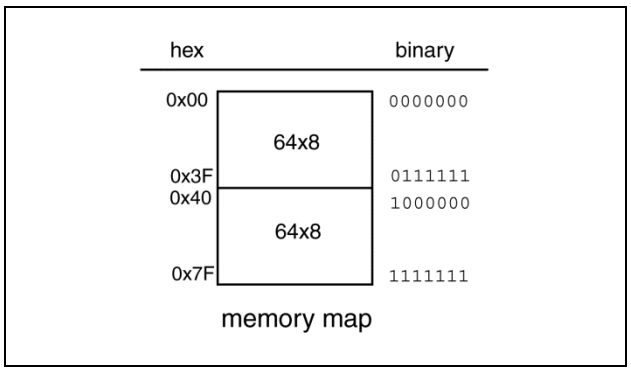


Figure 8-2: The final solution to Example 8-1.

Example 8-2: Memory Mapping with Four Devices

Show the address ranges in both hexadecimal and binary associated with the use of four 8Kx16 memories to form one 32Kx16 memory.

Solution: The good news is that this problem is similar to Example 8-1, only the numbers have changed to protect the innocent. The first step in this problem is to draw a diagram to help up see what exactly the problem

is asking us to do. Figure 8-3 shows a block diagram that is modeling four 8Kx16 individual memories as one 32Kx16 memory.

Our next step is to examine Table 8.1 and figure out some of the metrics we'll need to use to complete this problem. First, an 8K memory requires 13 address lines while a 32K memory requires 15 address lines. What this tells us is that the most significant two bits in the 15-bit address are what we'll need to use to differentiate between the least significant 13 bits. Also from Table 8.1 is the notion that the 13-bit range for an individual memory is [0x0000,0x1FFF]. We've seen this problem before; each 8Kx16 chunk of memory has a different upper two bit; these bits are "00", "01", "10", and "11". Yep, it's that binary sequence yet again. We complete this problem by pasting this set of bits in front of the address range values given for the 8K memory: [0x0000,0x1FFF]. Figure 8-4 shows the final solution to this problem with gory details included.

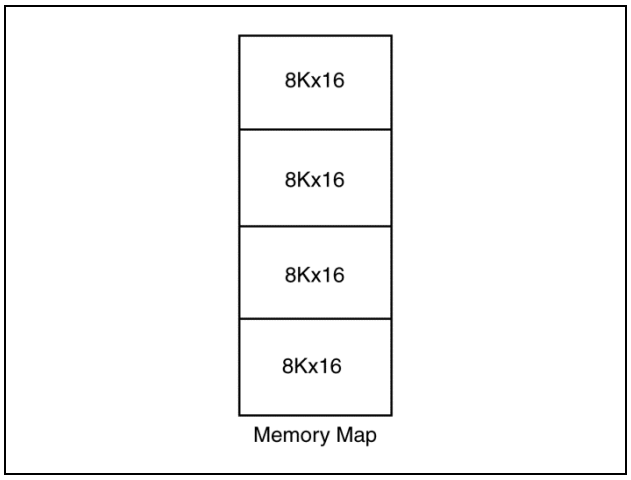


Figure 8-3: The diagram associated with Example 8-2.

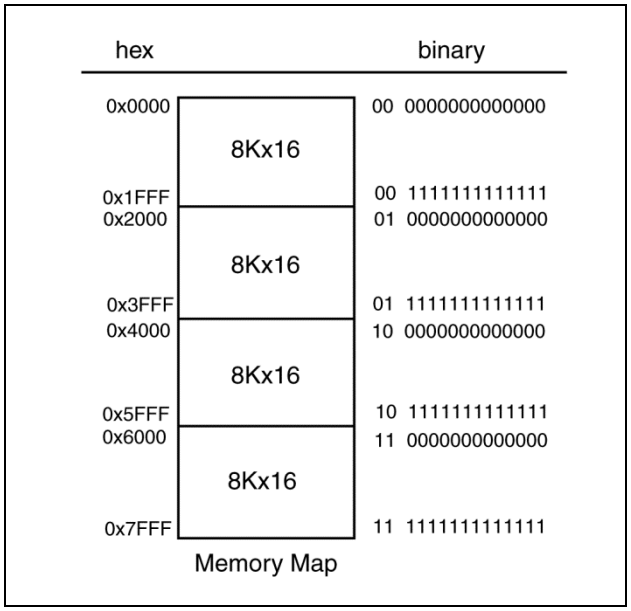


Figure 8-4: The final solution to Example 8-2.

8.3 Memory Organization

Discrete memory devices do not come in every possible capacity. This means if you need some type of special capacity, you're going to need to synthesize it from various memories of smaller capacities. The architecture of the overall memory created from smaller memories is what we refer to as memory organization. Additionally, if some digital system you are working with has a specified memory capacity but it is created from many smaller memories, you immediately know the capacity of the memory but you don't necessarily know any of the details regarding the organization of memory.

The most appropriate title for this section would be something like: "using many smaller memories to create a larger memory", but that title would run across two lines. The choice of "memory organization" attempts to reflect the notion that many times you'll need to satisfy your particular memory needs by configuring many smaller memory devices in a system such that it creates some particular form of a larger memory. You could use many different approaches to doing this; this section outlines only one major approach.

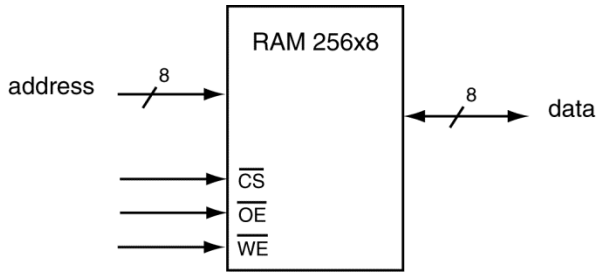
If you need to create a larger capacity memory from a many smaller capacity memories, you can do it in two basic ways: 1) increase the number of words that the memory can store, or 2) you can increase the effective width of the words stored in that memory. You could also do both, but we'll not deal with such problems in this section. The two subsections within this section deal with increasing word size and increasing the number of words store.

8.3.1 Extending Memory Word Length

Extending word length is the most straightforward approach to building larger capacity memories. The fun starts when we extend the number of addressable words in the next subsection. Extending word length is more straightforward because it only requires special circuit configuration and but typically does not require additional circuitry as does extended the addressable memory space. The best approach to explaining the concept of extending word length is through an example problem.

Example 8-3: Extending Memory Word Length

Show a circuit diagram that uses two of the following listed 256x8 RAMs to effectively create one memory with a 256x16 capacity. Assume the memory has bi-direction data lines. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



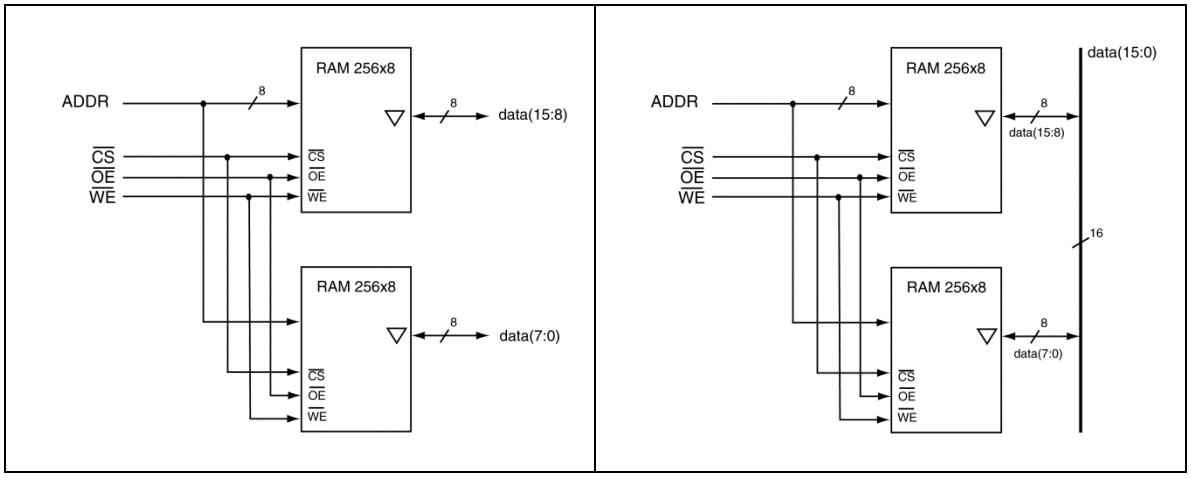
Solution: This problem is straightforward because we don't need to tweak the address lines in order to address all the possible words in the 256x16 memory. Figure 8-5(a) shows one possible architecture for the solution to this example. Here are the worthy things to note from the solution.

- The two 256x8 RAM devices share all the same control lines as both RAMs always need to act simultaneously for both reads and writes.

- The final circuit comprises of the two 256x8 RAM sharing the address lines. This is possible because the overall number of words for the larger capacity memory does not change.

- The meat of the solution lies in the interpretation of the outputs of the two 256x8 RAMs. The output of each 256x8 RAM becomes half of the final data width for the 256x16 RAM. In other words, each 256x8 RAM contributes eight bits to the final 16-bit output of the 256x16 RAM.

- The schematics in Figure 8-5(a) and Figure 8-5(b) are functionally equivalent. Sometimes it is clearer or easier to draw the schematic one way rather than the other.



(a) (b)

Figure 8-5: Two different memory configurations: (a) 512 x 8, and (b) 256 x 16.

8.3.2 Extending Memory Address Space

Although using multiple memories to extend the data width of an aggregate memory is straightforward, using multiple memories to extend the overall address range can be slightly tricky. This section describes these issues and provides some options for solutions.

Figure 8-6 highlights the main issue involved with extending address space in a multiple memory system. For this problem, we don't consider extending the data width. As you can see from Figure 8-6, the issues lie in how to handle the addressing needs of the individual memories. Figure 8-6 shows four MxN memories; we intend to include these four memories into one system. The resulting memory space is thus 4MxN. In order to have sufficient address space to address the 4MxN overall memory, we must increase the number of address lines on the system by two. In this way, the two extra address bits are sufficient to address one of the four internal memories.

One important thing to notice about Figure 8-6 is that the individual memories have tri-stated outputs. The characteristic helps define the overall problem: When we present the memory system with an "M+2" address, we need only one of the internal memories to drive their data onto the data lines. We somehow need the "M+2" address lines to effectively apply an address that addresses the full memory space but only actuates one of the internal memories. We only want one internal memory activated because the internal memories are sharing one set of data lines.

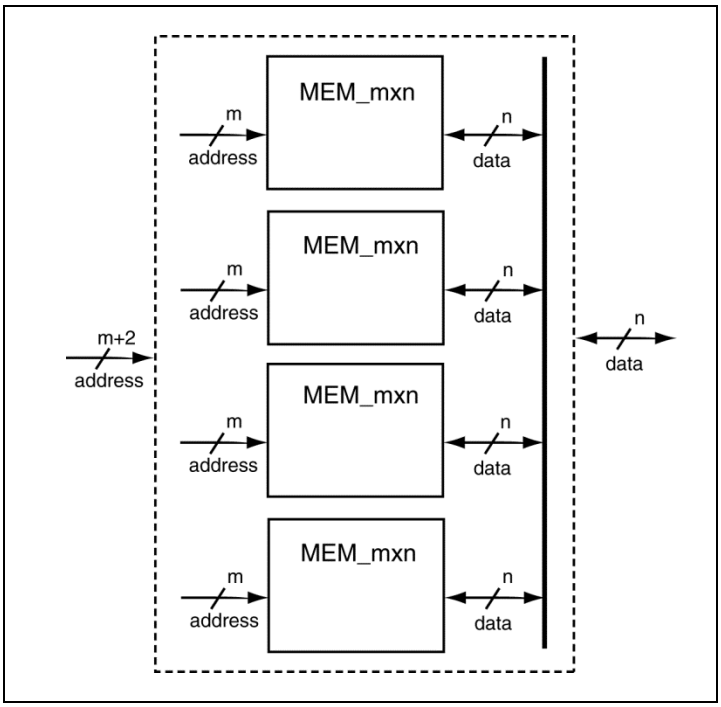


Figure 8-6: An overview of the extending address space dilemma.

Figure 8-7 facetiously shows an overview of our approach to the extending memory space. The approach we'll take is to insert circuitry into the "Magic_CKT" module. This module will then be responsible for translating the full "M+2" address space into m address lines and an appropriate number of control lines. We'll of course need at least two control lines, but there could be more depending on the control requirements of the internal memories. We'll present some relatively straightforward examples highlighting an approach based on relatively simple control requirements of the internal memories.

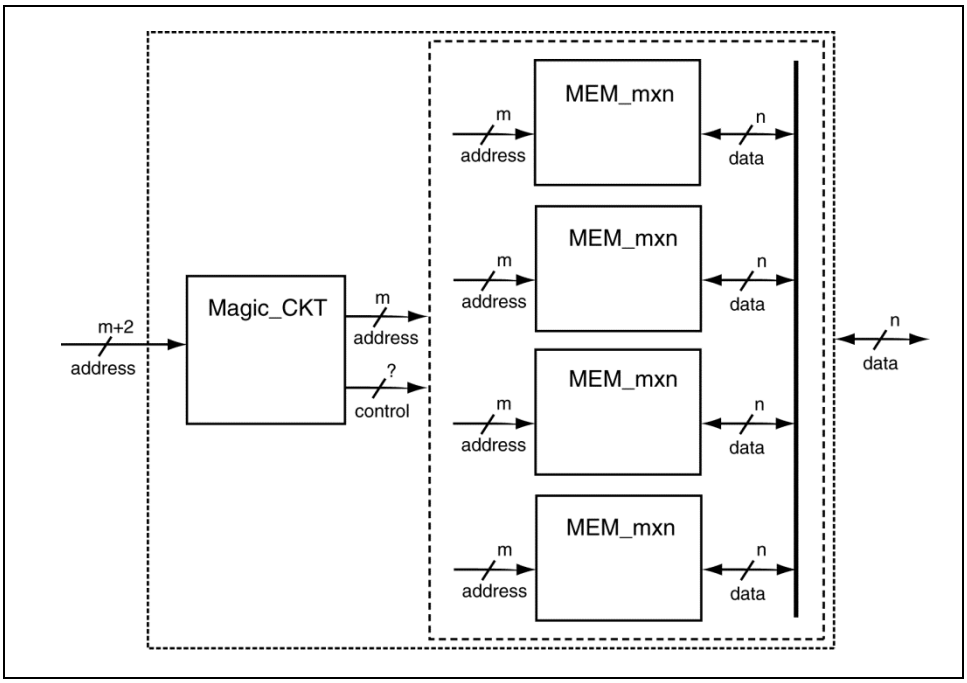
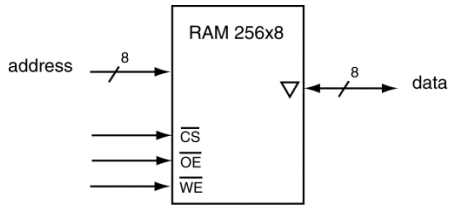


Figure 8-7: An overview of the solution to the extending address space dilemma.

Example 8-4: Extending Memory Address Space

Use as many of the following RAMs as you need to create a memory with a 512x8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



Solution: Figure 8-8 shows the first part of the solution to this problem, which is to draw a black box diagram of the final solution. In order to create an address space that accesses 512 memory locations, we need nine address lines. We gathered this information from Table 8.1. Since the underlying memories can address 256 memory locations, we'll need two of these memories to access the required 512 locations.

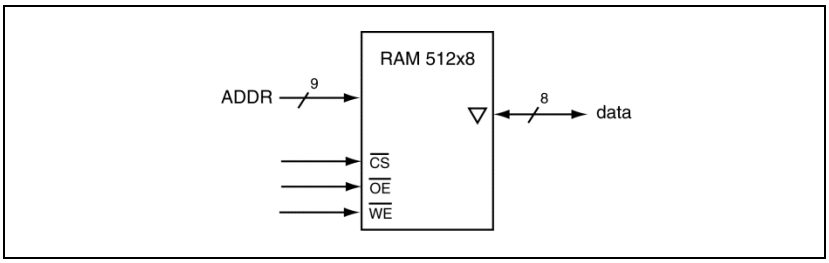


Figure 8-8: The black box diagram for the solution to Example 8-4.

Relative to Figure 8-7, we'll need to add circuitry that divides the address lines between the lines that are common to the each 256x8 device and the extra lines required for the larger memory configuration. For this problem, we'll have eight address lines for the underlying memory devices and one extra address line to differentiate between the underlying memories based solely on the "8+1" address lines. The approach we'll take is to insert a standard 1:2 decoder to handle the extra address line. Standard decoders are ideal devices for this application as they have only one active output at any given time. Figure 8-9 shows the final circuit solution to this problem. Here is some happy information regarding the solution in Figure 8-9.

- The decoder in Figure 8-9 is a standard 1:2 decoder with active-low outputs. We chose a decoder with active-low output in order to have those outputs properly interface with the active-low CS inputs of the individual memory devices. This standard decoder has only one active output a time; being that the outputs are active-low, we can consider the outputs of the decoder as "one-cold"[3]. This configuration provides the controls to enable only one memory device at a time. The input to the 1:2 decoder thus becomes the most significant bit in the overall 9-bit address. When the input to the decoder is '0', the decoder actuates the top memory; when the input to the decoder is '1', the decoder actuates the lower memory. Recall from the problem statement that when the memory's chip select is not active, the device effectively provides high-impedance to the data lines.

- The two memory devices share the lower eight address lines. Once again, the CS signal determines which memory device is active based on the ninth address line (the input to the decoder).

- The two memory devices share the OE and WE lines. The notion here is that some outside device will utilized these controls as required. There are generally no loading issues associated with these signals as only one memory device is actuated at a given time.

- The total number of address lines is independent of the size of the standard decoder. The characteristic that determines the minimum size of the standard decoder is the number of memory devices internal to the overall memory system. In other words, the standard decoder's responsibility is to use the appropriate address bit(s) to actuate the proper memory associated with all the address lines.

[3] For example, a decoder with one-cold outputs has one output at a '0' state and all other outputs at a '1' state.

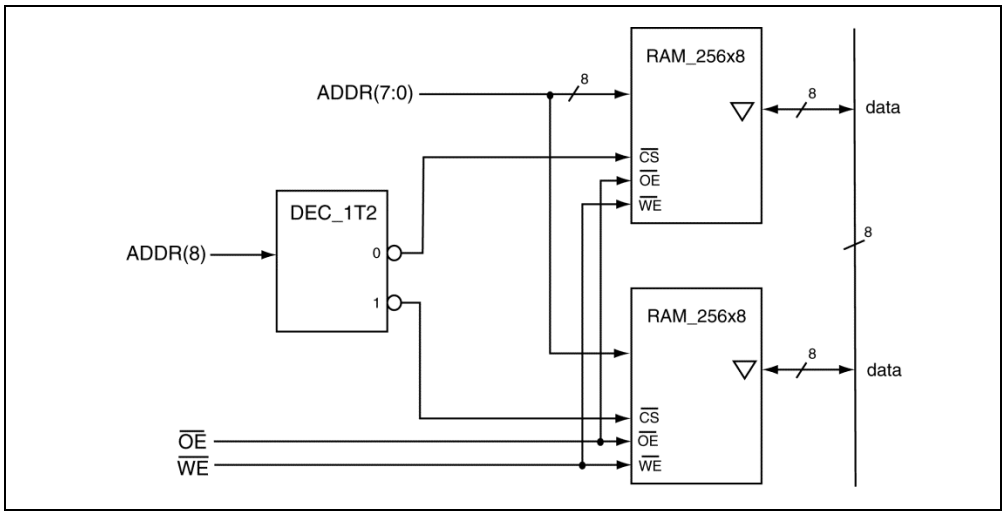


Figure 8-9: The final circuit solution to Example 8-4.

Let's take a closer look at the configuration. The final part of this solution is to generate of memory map that shows the overall memory space as well as the addresses associated with the underlying memory modules. The solution to this part of the problem is similar to the memory space discussion of section 8.2. Recall that we actively accessed information from Table 8.1 in those types of problems. Figure 8-10 provides the final solution to this example with some supporting notes to follow.

- In accordance with Table 8.1, the nine-bit addresses for the overall device start at all 0's and end with 0x1FF, or all 1's.

- Figure 8-10 shows the addresses of the underlying memories in both hexadecimal and binary. Note that the binary addresses have a space inserted in the number to highlight the notion that the ninth address bit is used to delineate between the two memories.

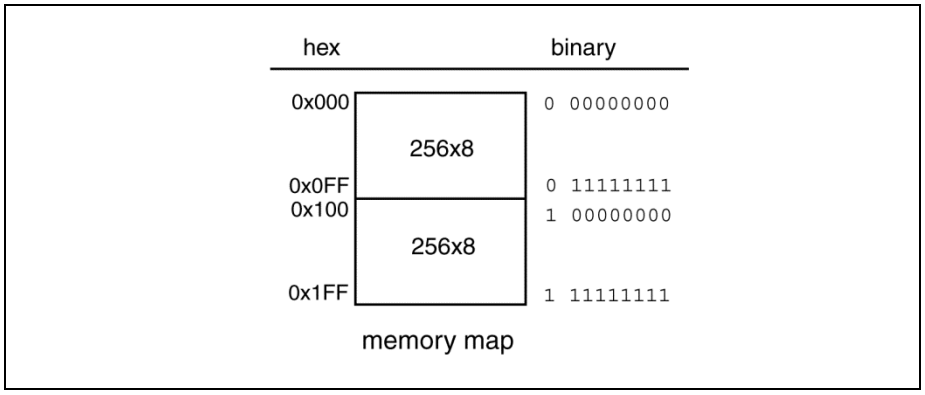
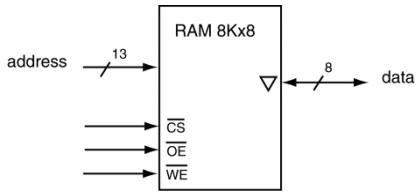


Figure 8-10: The memory map associated with Example 8-4.

Example 8-5: Extending Memory Address Space with More Devices

Use as many of the following RAMs as you need to create a memory with a 32Kx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



Solution: The first thing to note is that this problem is very similar to the previous example problem. That being the case, the first step in this problem is to draw a diagram of the final high-level object. Figure 8-11 shows the high-level schematic diagram associated with this problem. The first thing we see is that we need four devices with 8K worth of memory space to create a memory with 32K addressable memory locations. This means that we'll need four 8Kx8 devices in the final circuit. Once again using Table 8.1 we see that a memory with 32K memory locations requires 15 address lines.

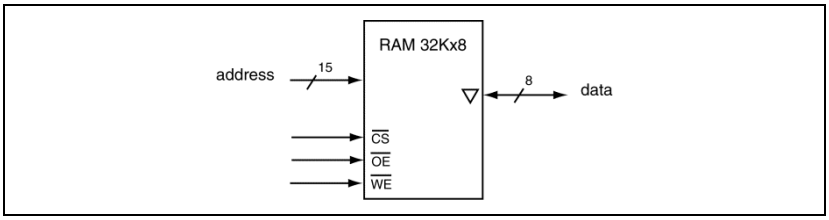


Figure 8-11: The black box diagram for the solution to Example 8-5.

The biggest similarity between this problem and the previous problem is with the use of a standard decoder to handle the "magic_ckt" in Figure 8-7. The standard decoder in this problem is slightly different in that it will need to choose between four different memories. This simply requires that the final circuit use a 2:4 standard decoder in place of the 1:2 decoder of the previous problem.

Figure 8-12 shows the final solution to the circuit portion of this problem. Note that this solution is similar in overall structure to the solution of Example 8-4, with the main difference being that we now need to choose between four discrete memory devices instead of the two devices. A standard 2:4 decoder easily handles this task by effectively using the two most significant bits of the 15-bit address lines as the select inputs to the standard decoder.

The memory map in Figure 8-13 shows the second part of this solution. Figure 8-13 highlights the mechanics of this solution most obviously in the binary numbers on the right side of the memory map. As you can see, each of the lower 13-bits for the individual memories are the same; only the two most significant memory bits differentiate the 15-bit address. As you would expect, since the circuit must choose between activating one of four memory devices, the circuit must use a minimum of two bits for this task.

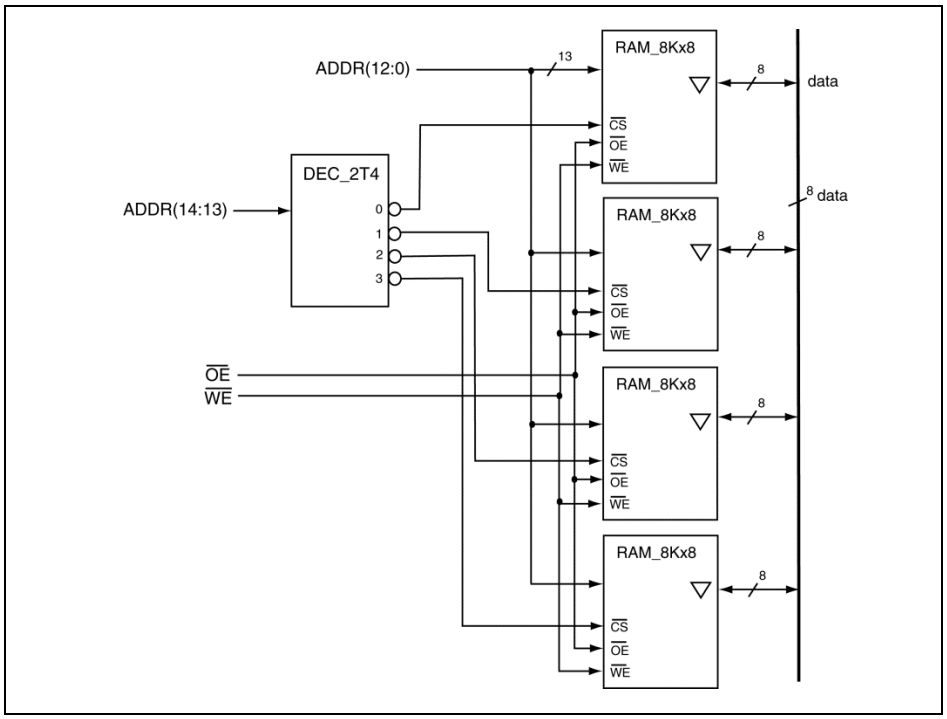


Figure 8-12: The circuit diagram solution for Example 8-5.

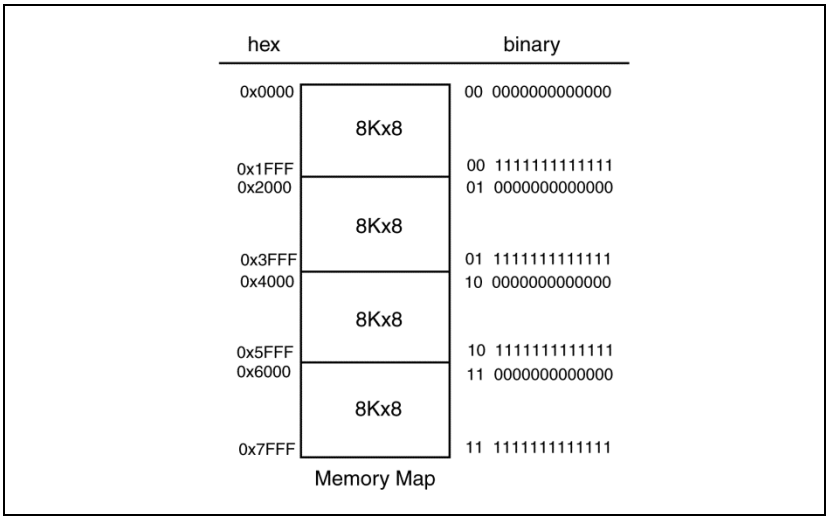
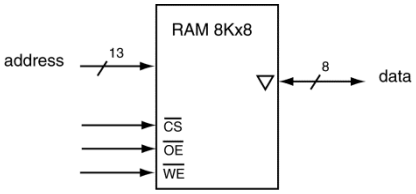
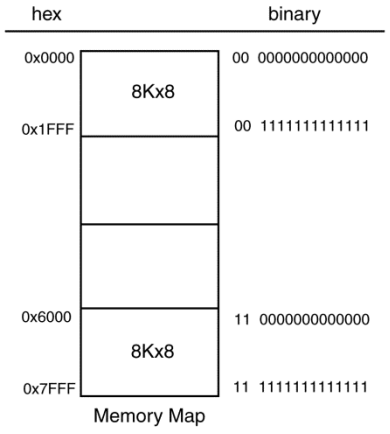


Figure 8-13: The memory map associated with Example 8-5.

Example 8-6: Circuit Design for a Memory Map

Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only 2:4 standard decoder, and 2) using only one 1:2 standard decoder. Use two of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.





Solution: This problem is slightly different in that it does not use the entire memory space listed in the problem description. In other words, the memory space has is divided into four sections, but only two of those sections contain active memory. Because there are only two sections with active memory, the solution only requires the use of a 1:2 standard decoder. We'll do this problem in two different ways in order to highlight the differences in using standard decoders of different size.

Figure 8-14 shows the most straightforward solution to this problem, which is to use a 2:4 decoder. The good thing about this solution is that each address in the 32K address space associated with the memory map is unique. The second part of this problem describes this issue further. The downside of the solution in Figure 8-14 is that the 2:4 standard decoder is partially unused. This implies the device is probably physically bigger than it needs to be, which may or may not be an issue[4]. Another possible upside of this solution is that is facilitates a later possible expansion of the memory map in that all the support hardware is in place; expansion would thus be a matter of dropping other 8Kx8 memory devices.

Figure 8-15 shows the solution for part 2) of Example 8-6. This solution replaces the 2:4 standard decoder from the part 1) solution with a 1:2 standard decoder. The upside of this solution is that it uses a smaller decoder. The possible downside of this solution is that each memory location in each 8Kx8 RAM is accessible using two different addresses. The problem results from effectively no longer using the ADDR(13) in the circuit solution. Because the solution is not using this address, the address bit is effectively a "don't care". As a result, the addresses of 0x1FFF and 0x17FF effectively access data from the same location in the lower-order 8Kx8 memory. In other words, for this solution, 0x1FFF and 0x17FF access memory location 0xFFF in the lower-order memory. While this certainly is an issue to consider, it may or may not be a problem for your particular system.

The main possible problem with this particular circuit design is that the memories may be driving the data bus at times where the memory is not being access. Recall that this is a relatively simple memory that uses the chip-select input signal to actuate the memory; when the memory is not actuated, the memory outputs are effectively in a high-impedance state.

[4] Keep in mind that if you plan on modeling the 2:4 decoder with VHDL and then synthesizing it, the synthesizer will probably mitigate the size issue of the standard decoder.

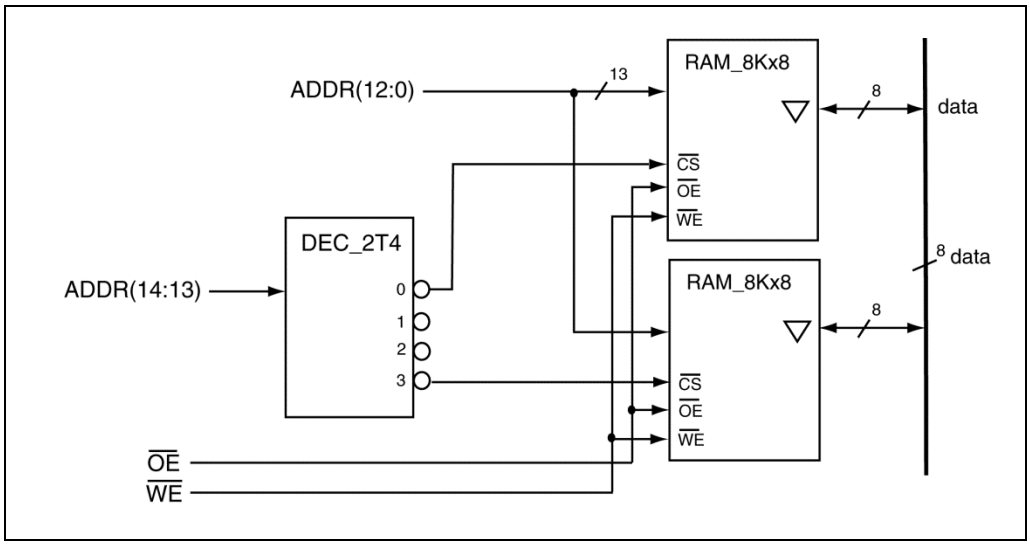


Figure 8-14: The black box diagram solution for solution for Example 8-6 part 1).

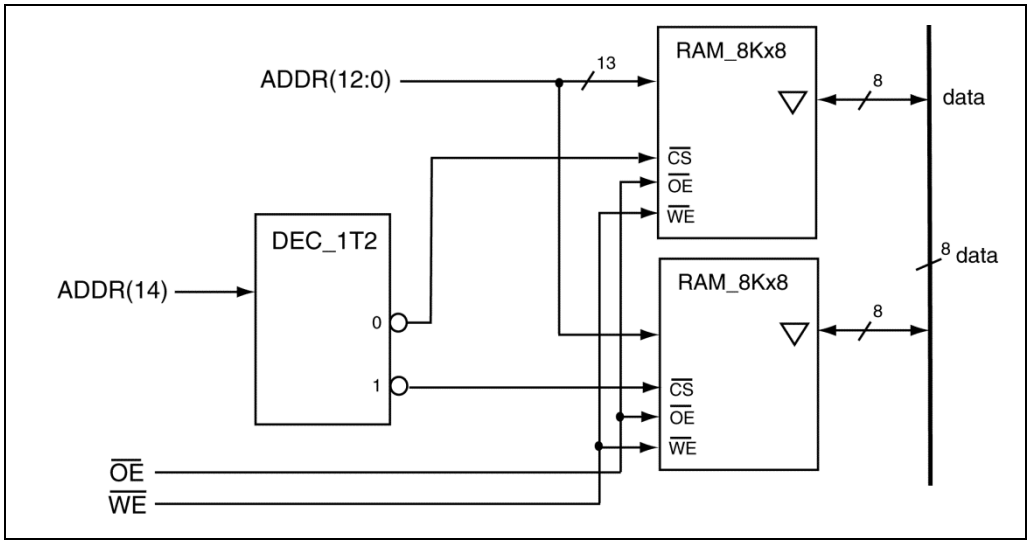
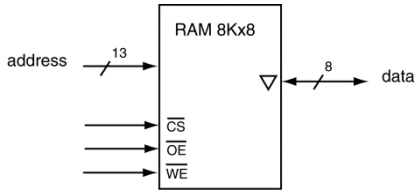
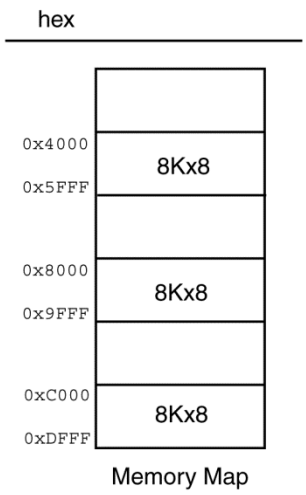


Figure 8-15: The black box diagram solution for Example 8-6 part 2).

Example 8-7: Using VHDL Modeling for a Memory Map

Design a circuit that implements the following memory map using both a 3:8 standard decoder and a by using a VHDL model to implement the "magic_CKT" portion of the circuit. Use three of the RAM devices listed below in your solution. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.





Solution: This problem presents a slightly different twist beyond the previous problems. The memory map is not full and the memories in the memory map are not contiguous. Neither of these characteristics are a big deal, they do present some interesting challenges. Thus, Figure 8-16 shows the first step to this solution, which is to draw the black box diagram of the high-level final circuit.

The most useful piece of information presented in Figure 8-16 is the notion that the circuit requires 16 address lines. We know this from examining the memory map in the problem statement. What we look for in problems like this the minimum number of address bits required to solve the problem. The worst case in the problem is the address with the most number of bits. In this problem, we can see that 0x4000 and 0x5FFF only require 14 bits, but every other listed address requires 16 bits. Thus, our final circuit requires 16 bits for the address lines. However, be sure to keep in mind that we won't be using every address in the 16-bit range, which is why we did not state any memory capacity information in the circuit diagram of Figure 8-16.

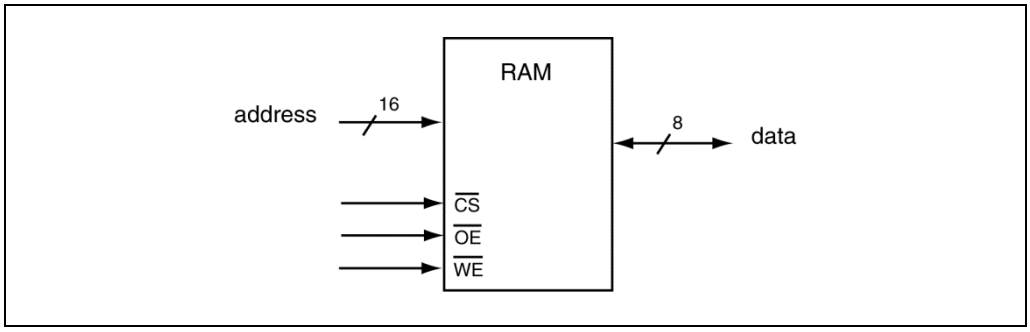


Figure 8-16: The black box diagram solution for Example 8-7.

The next part of this solution is broken into two parts. We need more information in order to complete the non-VHDL part, so let's get that out of the way right now. The best approach to first list the all the address ranges in both hex and binary for both the segments that include memory as well as the segments that are not associated with a memory device. The 8K address space requires 13 bit; the "magic_CKT" portion of the circuit handles the other three bits. In all likelihood, we'll be able to handle the "magic_CKT" portion of the circuit using a 3:8 standard decoder as stated in the problem. Figure 8-17 shows the result of this step.

Figure 8-17 conveniently shows the 16-bit addresses divided into 3-bit and 13 bit segments in the binary address listing. From this listing, you can see that the hex address values from the problem statement fall on 8K boundaries. This is good news as this will allow us to easily use the 3:8 standard decoder in the solution. The important information in Figure 8-17 includes an accounting for every address in the 16-bit address space. Note that a 16-bit address space is associated with a 64K memory. Since we're implementing this memory with eight 8K memory device, there should rightly be eight 8K segments in Figure 8-17. The truth is that some of unused segments represent more than one 8K with of address.

For example, the first segment listed in Figure 8-17 has no associated memory. This address range scans two 8K segments worth of memory, a fact that Figure 8-17 does not explicitly show. In reality, the first segment in the first address range covers 0x0000 through 0x1FFF while the second 8K segment covers 0x2000 through 0x3FFF. The other two ranges that do not have memory only cover one 8K segment.

The important thing to note in Figure 8-17 is that in the binary numbers associated with the memory, the range of the least significant 13 bits of the address is always 0x0000 to 0x1FFF. This means that the most significant upper three bits differentiate between the 8K memory spaces. We know that a 64K memory can be created from eight 8K memories; in this problem, we're only interested in the 64K space while the total memory capacity is only 24K. The most significant three bits in the binary address of Figure 8-17 then become the inputs to the associated 3:8 standard decoder. Without much hoopla, Figure 8-18 shows the final solution to the first portion of the problem.

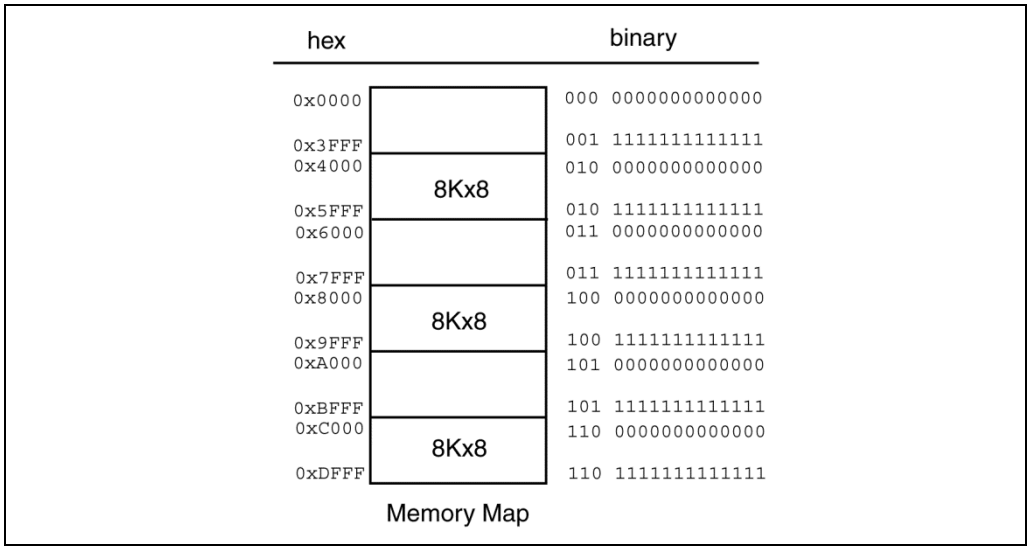


Figure 8-17: The full memory map (all addresses listed) for Example 8-7.

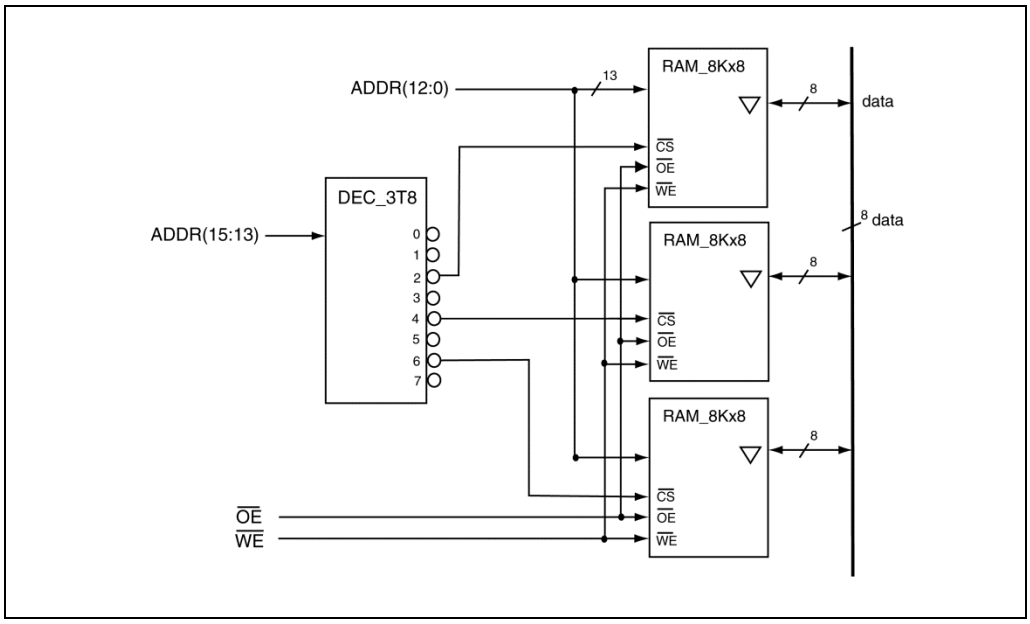


Figure 8-18: The black box diagram solution for first part of Example 8-7.

The potential problem with the circuit solution in Figure 8-18 is the notion that the 3:8 standard decoder is vast overkill for the problem. The problem could have used a 2:4 standard decoder since it only needed to control three memories. On the other hand, if we had used a 3:8 decoder, it would have required extra circuitry in addition to the decoder in order to make the address actuate the correct memories. The second part of this problem somewhat solves this issue by requesting you to provide a VHDL model that serves the same purpose as the 3:8 standard decoder.

Figure 8-19 shows the block diagram for the solution to the second part of Example 8-7. You can see from this diagram that our mission is to design a circuit that implements the "GEN_DEC" portion of the circuit. As the name of the box implies, our solution will simply be a generic decoder. This decoder will have three inputs and three outputs; the inputs are the most significant address lines while the outputs will actuate the appropriate discrete memory when the 16-bit address conditions are correct.

The solution to the second portion of the problem is approaching trivial once you realize all two things. First, the solution is a generic decoder, which is no big deal to model in VHDL. Second, the information presented in Figure 8-18 provides us with all the information we need to create the required generic decoder. Note that in Figure 8-18, the standard decoder actuates on output only when the inputs are "010", "100", and "110". More specifically, "010" actuates CS2, "100" actuates CS4, and "110" actuates CS6. If none of these three addresses is selected on the input lines, the module's outputs turns off all the discrete memory devices.

Figure 8-20 shows the VHDL model for the gen_dec modules. The only thing about this relatively simple model worth mentioning is to remind you that that discrete memory devices required active-low chip select inputs. Also somewhat worthy of note is the fact that the module's output is in one-cold form. As you can see, we are once again saved with the simplicity of model generic decoders in VHDL.

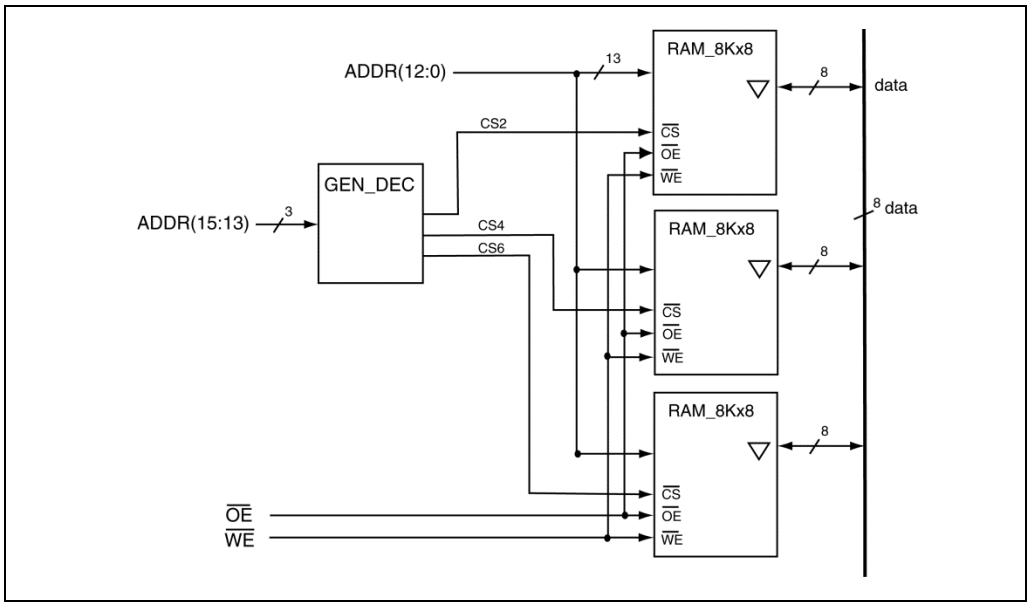


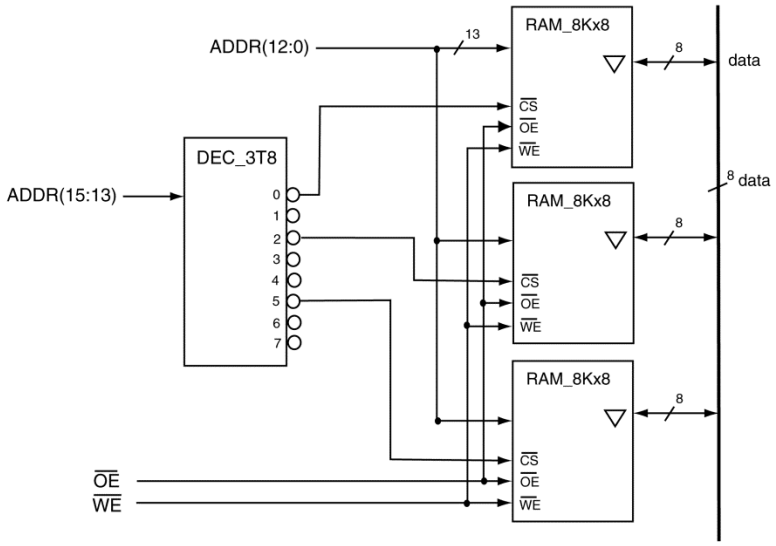
Figure 8-19: The black box diagram for the second part of the solution to Example 8-7.

```
entity gen_dec is
    Port ( ADDR_15_13 : in  STD_LOGIC_VECTOR (2 downto 0);
           CS2 : out  STD_LOGIC;
           CS4 : out  STD_LOGIC;
           CS6 : out  STD_LOGIC);
end gen_dec;

architecture gen_dec of gen_dec is

begin

   my_dec: process (ADDR_15_13)
   begin
      case ADDR_15_13 is
          when "010"  => CS2 <= '0'; CS4 <= '1'; CS6 <= '0';  -- 2
          when "100"  => CS2 <= '1'; CS4 <= '0'; CS6 <= '0';  -- 4
          when "110"  => CS2 <= '1'; CS4 <= '1'; CS6 <= '0';  -- 6
          when others => CS2 <= '1'; CS4 <= '1'; CS6 <= '1';  -- catchall
      end case;
   end process my_dec;

end gen_dec;
```

Figure 8-20: The VHDL model of the "GEN_DEC" module of Figure 8-19.

Example 8-8: Generating a Memory Map From a Circuit

Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



Solution: This problem is the same old memory mapping problem but in the reverse order. This problem provides a circuit and you are responsible for generating a memory map. The first thing to notice about this problem is that there are three 8K memories in the circuit. The second thing to notice is that the problem uses a standard 3:8 decoder for the "magic_CKT" part of the circuit. Thus, the inputs to the standard decoder are effectively the three most significant address bits (15:13), which implies the circuit can possibly address up to a 16-bit address space. The problem only uses 3/8 of the total possible address space, or 24K.

Each 8K RAM shares the same 13 bits of address lines. This means the address range for any 8K RAM by itself is 0x0000→0x1FFF, with the standard decoder handling the other three bits. The best approach to take for this problem is to note that the 64K address space is divided in to eight 8K segments, but the circuit uses only three of the possible eight segments. The first used segment is associated with the three most significant address bits of "000" as indicated by that RAM's chip select being connected to the '0' output of the standard decoder. This makes the range for the first segment 0x0000→0x1FFF. The next 8K segment is not used; the starting address of this unused segment is one greater than the last valid address associated with the previous segment, or 0x2000. The ending address of this unused segment is 0x1FFF greater than 0x2000, or 0x3FFF.

You can continue this same type of analysis for all the other segments in the problem. The only slightly tricky thing to note is that the segments associated with 3-4 and 6-7 represent two 8K segments, which means the range effectively has 16K worth of segment space, or 0x0000→0x3FFF. After you realize all of this, the problem becomes somewhat of a math problem. Figure 8-21 shows the final solution to this problem with the addresses listed in both hexadecimal and binary for your viewing convenience.

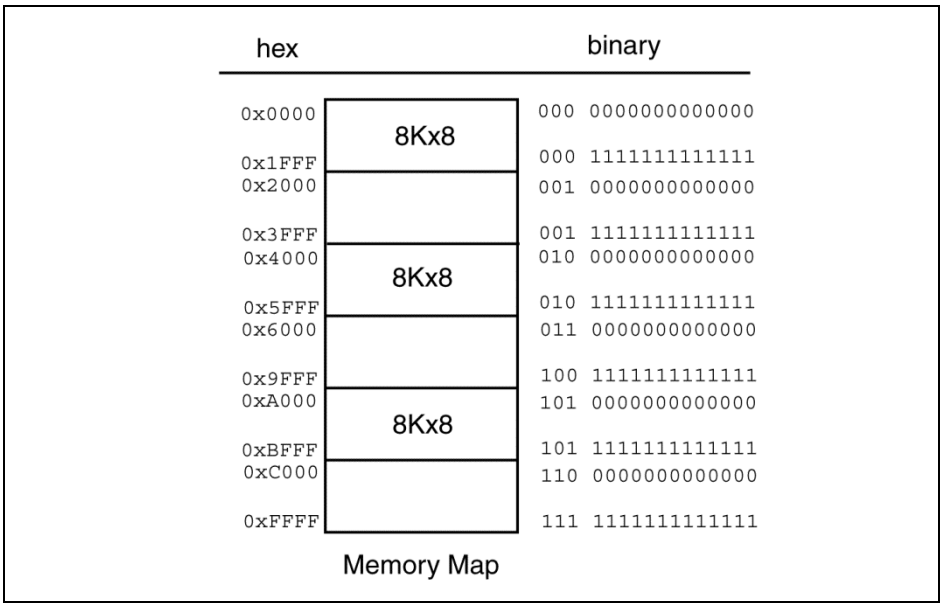


Figure 8-21: The final memory map for Example 8-8.

8.4 VHDL Models for Structured Memory

Although there is a lot to say about modeling, structured memory using VHDL, this section simply aims to get you started in the area. For further information, be sure to check out other VHDL resources. This section attempts to describe basic structured memory models in terms that you're already familiar with, thus, the models presented in this section by no mean represent the "best way" to model structured memory.

8.4.1 Generic ROM VHDL Model

Probably the simplest structured memory device to model using VHDL is an asynchronous ROM. For this section, we'll show and describe a model for 32x4 asynchronous ROM. The notion of asynchronicity relates to the fact that this ROM does not contain a clock input. As you'll see in later synchronous memory models, some memory have clock inputs, which the memory uses to synchronize its various operations. Figure 8-22 shows the black box model for the simple ROM we'll be modeling in this section. Figure 8-23 shows the VHDL model implementing the ROM in Figure 8-22.

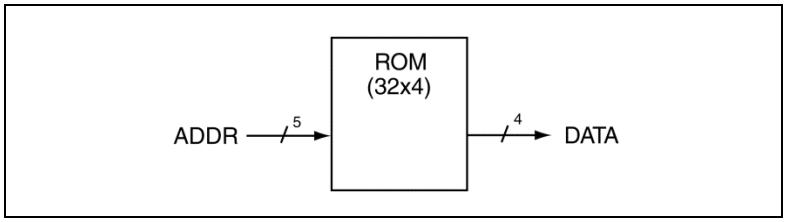


Figure 8-22: The black box diagram of a 32x4 RAM.

```
entity gen_async_rom is
    Port ( ADDR : in  STD_LOGIC_VECTOR(4 downto 0);
           DATA : out STD_LOGIC_VECTOR(3 downto 0));
end gen_async_rom;

architecture gen_async_rom of gen_async_rom is

   ---------------------------------------------------------------------------(1)
   TYPE vector_array is ARRAY(0 to 31) of STD_LOGIC_VECTOR(3 downto 0);

   ---------------------------------------------------------------------------(2)
   CONSTANT my_memory: vector_array := ( X"0", X"1", X"2", X"3",
                                         X"4", X"5", X"6", X"7",
                                         X"8", X"9", X"A", X"B",
                                         X"C", X"D", X"E", X"F",
                                         X"F", X"E", X"D", X"C",
                                         X"B", X"A", X"9", X"8",
                                         X"7", X"6", X"5", X"4",
                                         X"3", X"2", X"1", X"F");

begin

   ---------------------------------------------------------------------------(3)
   DATA <= my_memory(CONV_INTEGER(ADDR));

end gen_async_rom;
```

Figure 8-23: The VHDL model of a 32x4 RAM.

Here are some details regarding the more interesting features of the ROM VHDL model of Figure 8-23. These details only include the VHDL details not covered in previous VHDL models. The parenthetical numbers below refer to the commented notes listed in the VHDL model of Figure 8-23.

(1) We typically define structured memory objects in VHDL using the TYPE and ARRAY keywords. The TYPE keyword allows users to define their own types, which is similar to the notion of defining "state types" in VHDL behavioral modeling of FSMs. This ROM labels its type as a "vector_array". The ARRAY keyword defines the number of words in the memory while the text after the "of" keywords defines both the type and width of each word. This ROM object defines a structure memory object that contains 32 words (as specified by the "0 to 31" metric). Each word is a STD_LOGIC_VECTOR type with each word having a four-bit width (as specified by the "3 downto 0").

(2) Now that we have defined a type for the ROM, we now must officially declare the ROM structure. Because the values in the ROM will never change, we declare the ROM object as a constant type using the CONSTANT VHDL keyword followed by the name ("my_memory") that we'll use to access the object. The declaration of this constant has two parts: the declarative part and the initialization part. Since this is a ROM, there will never be a need to write to this memory. This being the case, we must preset, or initialize, the values into the ROM. We chose to use hex notation to initialize the ROM in an effort to save space and prevent a visual barrage of 1's and 0's. Note that the definition provides 32 initialization values as the ROM expects. Note that this definition is no different from the definition of signals, which specify a type, a label, and initialization values. Lastly, this form of initialization for the ROM works great for simulation; actual initialization of a ROM is a deep subject that has specifics that are all over the map, so we'll not get into anything here.

(3) Accessing data in the ROM is a matter of providing the ROM with the address of the word we want to access. In this case, the five-bit ADDR lines in the object provide access to 32 different words in memory. The ROM object uses the ADDR lines as an index into memory. The important thing to note here is that ARRAY types in VHDL can only be indexed using integers. Because ADDR is a STD_LOGIC type, we must convert that value to an integer. VHDL has many type conversion

functions in its various libraries; the **`CONV_INTEGER()`** function serves to convert its STD_LOGIC type to an integer, so you can use it to access the words in the ROM.

Once you have officially defined the ROM, it is ready for use. The notion here is that this ROM is asynchronous, which means nothing about the ROM synchronizes to a clock edge of a control signal. The notion of using this ROM in a circuit entails two steps. First, give the ROM valid data on the address lines (ADDR) of the word in ROM you desire the data for. Second, you wait a finite amount of time before the data is officially ready on the DATA outputs of the ROM. The amount of time you need to wait is dependent upon how the VHDL synthesizer creates the ROM[5].

8.4.2 Generic RAM VHDL Model

Modeling a synchronous RAM is the next step in our journey of modeling structured memory objects with VHDL. Figure 8-24 shows the RAM we'll describe in this section. As you can see from examining schematic diagram in Figure 8-24, this RAM supports some form of synchronicity as is evident from the presence of the CLK input. You would not know what part of the RAM is synchronous unless somebody told you or you were to examine the associated VHDL model. Lucky for us, Figure 8-25 shows the associated VHDL model for the RAM.

What you can determine from Figure 8-24 is that the RAM 's capacity is 32 words with each word being four bits wide. You can determine this in two ways. First, the RAM in Figure 8-24 clearly states this in the parenthetical expression in the RAM black box itself. The other method you can use to determine the capacity is that the ADDR lines are five-bits wide, which indicates that this RAM can address 2^5, or 32 different words. The width of the words is four bits as indicated by the width of both the D_IN (input) and D_OUT (output) lines. The width of the input and output data lines should always agree in memory such as RAMs.

As you see from examining Figure 8-25, the RAM model is very similar to the ROM model of Figure 8-23. Thus, you can rather think of a RAM as simply a ROM that you can write to. There are a few new items present in the VHDL model for the RAM that is worth further description.

(1) The RAM model uses a process to handle writing to the RAM. Figure 8-24 shows that this RAM has two control inputs: CLK and WE, which handle the writing of data to the RAM. In order to place new data into the RAM (write), the WE input (write enable) must be asserted at the when a rising edge of the CLK input occurs. In this way, writes to the RAM are synchronous, meaning they are synchronized to the active edge of the CLK input. When conditions on the control inputs are correct, data on the D_IN lines overwrites data at the RAM address specified by the ADDR lines. The notion of "overwriting" implies that the data previous at the given location is forever lost.

(2) Despite our claim that this RAM is synchronous, the truth is that only write operations to the RAM are synchronous. The method used to handles reads is similar to read operations in the ROM. It's good to note here that RAM reads are not synchronous as the code indicates by disassociating the read modeling from any notion of the CLK signal.

[5] Creation of structured memory in VHDL depends upon many factors. The biggest factor is the resources available on the target device where the ROM will reside. We'll leave those details to your own specific needs.

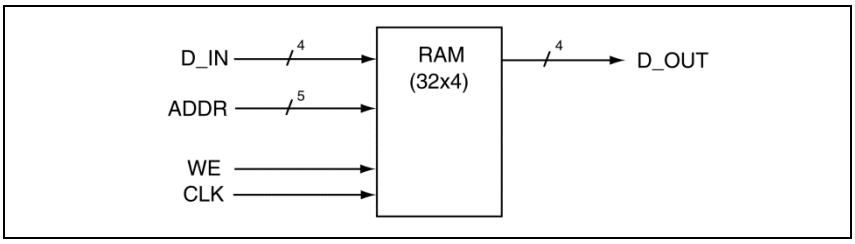


Figure 8-24: The schematic diagram of a 32x4 RAM.

```
entity gen_ram is
port (    CLK : in std_logic;
           WE : in std_logic;
         ADDR : in std_logic_vector(4 downto 0);
         D_IN : in std_logic_vector(3 downto 0);
        D_OUT : out std_logic_vector(3 downto 0));
end gen_ram;

architecture gen_ram of gen_ram is

   type ram_type is array (0 to 31) of std_logic_vector (3 downto 0);
   signal gen_ram : ram_type;

begin
   ram_write: process (CLK,WE)
   begin
      --- writes to RAM
      ----------------------------------------------------(1)
      if (WE = '1') then
         if (rising_edge(CLK)) then
            gen_ram(conv_integer(ADDR)) <= D_IN;
         end if;
      end if;

   end process ram_write;

   --- reads from RAM
   ----------------------------------------------------(2)
   D_OUT <= gen_ram(conv_integer(ADDR));

end gen_ram;
```

Figure 8-25: The VHDL model of a 256x8 RAM.

That's all we'll say about this RAM. The moral of this story may be that when you hear statements such as "synchronous RAM", you really can't be sure of what exactly that means. To get the full story, you must either read the associated datasheet or examine the VHDL model.

8.4.3 Generic Bi-Directional RAM VHDL Model

One characteristic of some memory devices is the notion that we refer to them as being bi-directional. This label generally means that the devices have only one set of "data" lines, which are used to output data (read) and input data (write). This sharing of the data lines is in an effort to save routing resources; as you may guess, you cannot both simultaneously write to and read from the device. There are many ways to model bi-directional memories; this section only describes one of those ways.

Figure 8-26 shows a schematic diagram of the bi-directional RAM we'll describe in this section. We use a doubly directed arrow[6] to indicate that the RAM is bi-directional. Additionally, the inverted triangle next to the bi-directional signal indicates that the device has some sort of tri-state capability. This device has three control

[6] An arrow with a head on each end. More exciting than you may have expected.

lines, one of which is a clock signal, which implies something about this device is synchronous. Once again, we'll have to read the associated datasheet or check the associated VHDL model in order to ascertain what is synchronous on this device.

In every case I've ever seen, the tri-stating and bi-directionality of the data lines indicates that the device is driving the data lines as a function of a read operation and is in a high-impedance state when the device is not driving the data lines. The OE signal, generally known as an "output enable" and officially drives data onto the data lines when it is asserted. When the OE signal is not asserted, the outputs are in a high-impedance state. Consequently, a write operation consists of the unasserting the OE signal and tweaking the other two signals. As we'll see by looking at the VHDL model, this device is similar to the RAM we examined in 8.4.2; the only difference is the notion of controlling the tri-state outputs on write operations.

Figure 8-27 shows the VHDL model associated with the schematic diagram of Figure 8-26. Here is yet more interesting stuff regarding the numbered comments in the associated VHDL model. As you'll see, the bi-directionality of this structured memory makes the VHDL model somewhat different from the previous memories we examined.

(1) We've opted to initialize this RAM using this particular statement. This statement sets all bit locations in the memory to '0'. Once again, this is something that makes the simulator happy but may not be synthesizable based on the actual flavor of RAM your circuit will actually use.

(2) The code uses one process to model both read and write aspects of the memory. This works well because we are either "reading" from the device, or "writing" to the device, but not both. The functionality described in the previous sentence is nicely modeled by the notion of an if statement. Recall that an if statement is sequential; if the if clause does not evaluate as true, the model executes the else clause. The if clause in Figure 8-27 handles the driving of the output data when the OE signal is asserted, which is the read operation.

(3) The else clause executes when the if condition does not evaluate as true.

(4) When the OE signal is not asserted, the output is driven to its high-impedance state. The previous sentence is another way of saying that the outputs "shut off" when the OE signal is unasserted. Under these conditions, there is thus no output. But, some other external device can drive the BI_DIR_DATA lines; when such a device is driving these lines, the RAM can officially perform a write of that data. This is possible because the model declares the BI_DIR_DATA lines as an "inout" type in the original entity declaration.

(5) When the else clause is evaluated, the RAM is not driving its data onto the data lines. There now is a possibility that the RAM can perform a write operation. From examining the VHDL code, we can see that writing to the RAM depends on the proper conditions of the CLK and WE control signal, in the same way as the non-bi-directional RAM device we previously described. Once again, when the WE (write enable) signal is asserted and there is a rising-clock edge on the CLK signal, the data driven to the data lines by some external device are written to the bi-directional RAM at the location specified by the ADDR lines. From the style of this code, we can see that the data write operations are synchronous while read operations are non-synchronous.

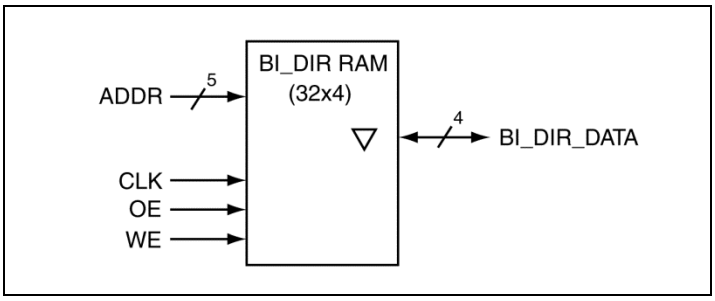


Figure 8-26: Schematic diagram of 32x4 bi-directional RAM.

```
entity bi_dir_ram is
    Port ( BI_DIR_DATA : inout STD_LOGIC_VECTOR (3 downto 0);
           ADDR : in STD_LOGIC_VECTOR (4 downto 0);
           OE : in STD_LOGIC;
           WE : in STD_LOGIC;
           CLK : in  STD_LOGIC);
end bi_dir_ram;

architecture bi_dir_ram of bi_dir_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);

   ------------------------------------------------------------(1)
   SIGNAL BD_RAM : memory := (others => (others =>'0') );
begin

   my_bi_dir: process(CLK,OE,WE,BI_DIR_DATA,ADDR,BD_RAM)
   begin
      ------------------------------------------------------------(2)
      if (OE = '1') then
         BI_DIR_DATA <= BD_RAM(conv_integer(ADDR));

      ------------------------------------------------------------(3)
      else
         ------------------------------------------------------------(4)
         BI_DIR_DATA <= (others => 'Z');

         ------------------------------------------------------------(5)
         if (WE = '1') then
            if (rising_edge(CLK)) then
                BD_RAM(conv_integer(ADDR)) <= BI_DIR_DATA;
            end if;
         end if;

      end if;
   end process my_bi_dir;

end bi_dir_ram;
```

Figure 8-27: The VHDL model of a 32x4 bi-directional RAM.

8.4.4 Generic Dual-Port RAM VHDL Model

As you can see by now, memory comes in many different flavors and we prefix with many different acronyms. The good news is that you can relatively easily model these flavors using VHDL. We conclude our study of VHDL RAM modeling with a specialized but relatively common type of RAMs that we refer to as a "dual-port RAM". Once again, when you hear the phrase "dual-port", you can't automatically know what exactly that is without perusing the datasheet or examining the VHDL model. The notion of "ports" on memory devices

generally applies to the either the input or output data lines, or both. The use of the word "port" somewhat refers to the inflow and/or outflow of data from the memory.

What you may not know yet is that computer use registers extensively to do the thing that computers do. As you probably know, a typical computer uses many different types of memory. The two main issues in computer memory are the capacity and speed of the memory. The range of memory in a typical computer starts with relatively small and fast memory and ends with relatively large and slow memory. The fastest memory on a typical computer is individual registers and the "register file"; the slower type of memory on a computer is something like the hard drive. The fastness of registers comes from the fact that it is semiconductor memory; the slowness of the hard drive comes from the fact the memory is magnetic and is only mechanically accessible. In summary, there are many differences between these types of memory but one of the main differences is the speed at which the CPU can access them.

When we refer to a "register file", we typically refer to a structured memory that we model as a bunch of individually named and accessible registers. We access the registers at the human level by name; some software device such as a compiler or assembler translates the names to an address. Thus, the register file is nothing more than a bunch of fast memory that the CPU can use. Register files come in many flavors; the one we'll examine in this section is typical of a structured memory (or a set of structured memories) connected in such a way as serving as to serve the needs of a computer's register file.

Most processors have a register file in which to store intermediate calculations. You can easily model this register file as a dual-port RAM. Figure 8-28 shows a schematic diagram of the dual-port RAM we'll examine in this section. As you can see from the schematic diagram, the dual-portness of this RAM means that there are two output ports (DX_OUT and DY_OUT). There seems to be two input ports in reference to the two different address values (ADRX and ADRY), but there is only one associated data port (D_IN). The only way we'll be able to figure this out is by looking at the VHDL model. Lastly, note that the schematic diagram contains two control signals: CLK and WE; we've seen those before.

Figure 8-29 shows the VHDL model associated with the schematic diagram of Figure 8-28. Although the name sounds almost as impressive as the schematic diagram appears[7], in reality, our model of the dual-port RAM is strikingly similar to our previous structured memory models. Here is a description of the new stuff found in Figure 8-29 referenced by the parenthetical comments in Figure 8-29.

(1) This RAM is very similar to the generic RAM we looked at earlier. The code handling the write process uses two control signals in the same way as we used them in the generic RAM with write operations controlled by the having the WE signal asserted at the same time as receiving a rising edge from the CLK signal (thus RAM writes are synchronous). The key to understanding this RAM is in this if statement in combination with the code appearing after the (3) commented line. Overall, it appears that there is one set of registers (32 of them), and we can only write one register at a time. This looks normal, so we'll save the punch line for our (3) comments.

(2) Despite the dual-portness of this RAM, we can only write one address at a time. Other than that, we've seen this flavor of write code previously. The important thing to note here is that in spite of the fact that this RAM has two address lines, we can only write to the RAM location indexed by the ADRX signal. Thus, in the context of writing, we do not use the ADRY signal.

(3) Finally, this portion of the code handles read operations associated with the RAM. This RAM has two output ports, which can effectively read two different memory locations as addressed by the ADRX and ADRY signals. This style of reading is the same was what you have seen previously.

[7] Whatever that means…

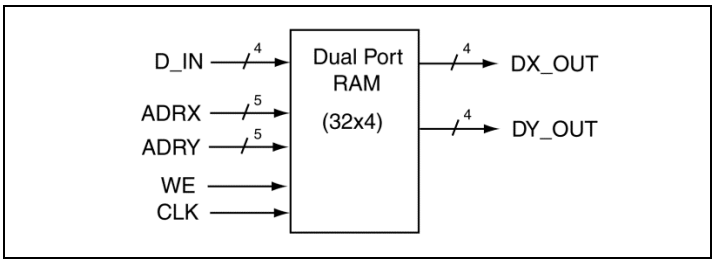


Figure 8-28: Schematic diagram of 32x4 Dual Port RAM.

```
entity gen_dual_port_ram is
    Port ( D_IN   : in  STD_LOGIC_VECTOR (3 downto 0);
           ADRX   : in  STD_LOGIC_VECTOR (4 downto 0);
           ADRY   : in  STD_LOGIC_VECTOR (4 downto 0);
           WE     : in  STD_LOGIC;
           CLK    : in  STD_LOGIC;
           DX_OUT : out STD_LOGIC_VECTOR (3 downto 0);
           DY_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end gen_dual_port_ram;

architecture gen_dual_port_ram of gen_dual_port_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL dp_ram: memory := (others=>(others=>'0'));

begin

   process(clk,WE)
   begin
      ----------------------------------------------------(1)
      if (WE = '1') then
         if (rising_edge(CLK)) then
            ------------------------------------------------(2)
            dp_ram(conv_integer(ADRX)) <= D_IN;
         end if;
      end if;
   end process;

   ----------------------------------------------------(3)
   DX_OUT <= dp_ram(conv_integer(ADRX));
   DY_OUT <= dp_ram(conv_integer(ADRY));

end gen_dual_port_ram;
```

Figure 8-29: Interior model of Dual Port RAM shown in Figure 8-28.

The code in Figure 8-29 does a great job of modeling the dual-port RAM at a high level. If you really intend on synthesizing the structured memory, it is useful to know how the synthesizer may handle this request for this flavor of dual-portness. The notion here is that the synthesizer is not going to invent anything new; it is instead going to attempt synthesize the dual-port RAM out of devices it is familiar with, such as generic RAMs.

We can model the dual-port RAM using two generic RAMs that have some special and clever connections. Figure 8-30 shows a low-level model of the dual-port RAM. The key to understanding how this device actually operates is to see that there are two separate sets of address lines and two separate sets of output lines, but there is only one WE line and most importantly, only one set of data lines. This certainly implies that you can perform simultaneous memory reads from two separate memory locations. But being that there is only one set of data lines, the best you can do is write the same data to two different memory locations. This may or may not float your boat, as we've been hinting all along; this is an excellent model for a typical computer's register file.

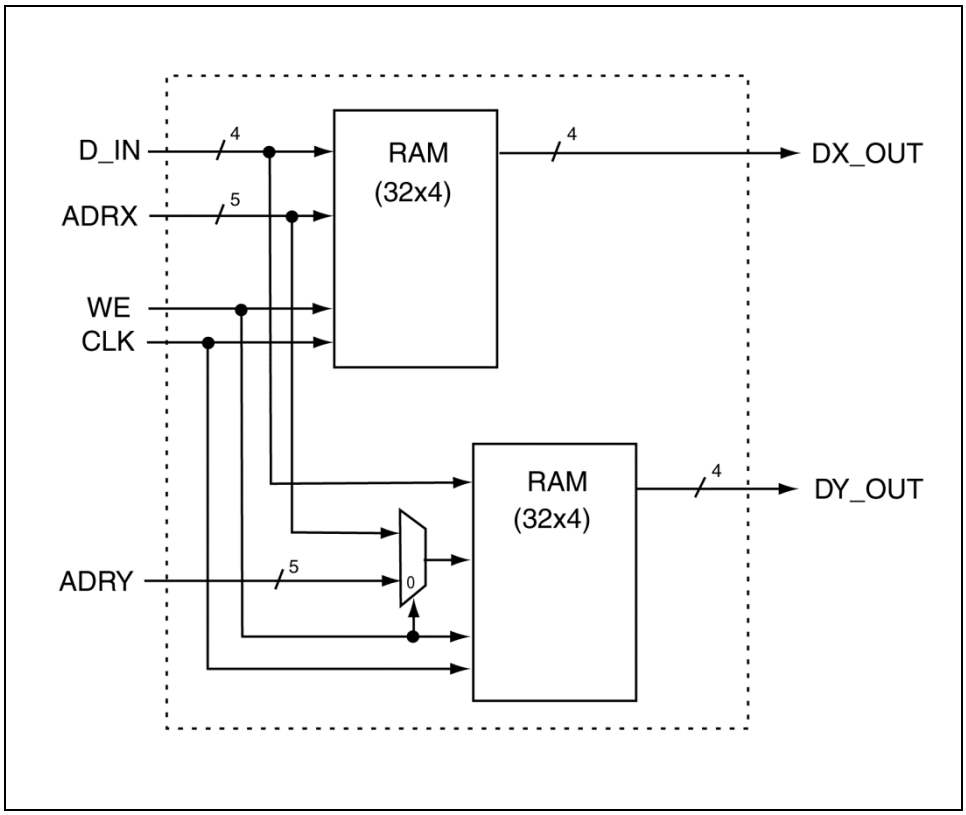


Figure 8-30: Interior model of the dual port RAM modeled in Figure 8-29.

8.5 Chapter Summary

- Memory systems are typically designed as many individual smaller memory units as opposed to one single larger memory unit. This form of design generally is more versatile and often is less expensive which means it is used quite often in digital design.

- The term memory mapping is typically associated the notion of using many smaller memories to create a larger memory. This larger memory does not need to be complete in terms of the full memory coverage of the associated address space. The memory map provides a visual indication of the starting and ending address of the smaller memories that may or may not be present in the design.

- Larger capacity memories can be created using many smaller memories in two different ways. The digital design can either extend the address space or extend the width of the associated data.

- Structured memories can be easily modeled using VHDL. The differences between VHDL models for ROMs, RAMs, bi-directional RAMs, and a dual-port RAMs are relatively minor.

8.6 Chapter Exercises

1) Show the address ranges in both binary and hexadecimal associated with the use of two 256x8 memories to form one 512x8 memory.

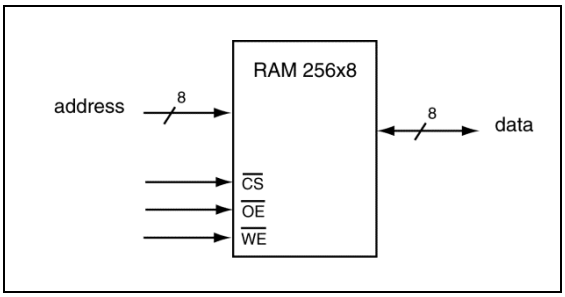
2) Show the address ranges in both binary and hexadecimal associated with the use of two 4Kx8 memories to form one 8Kx8 memory.

3) Show the address ranges in both binary and hexadecimal associated with the use of four 4Kx8 memories to form one 16Kx8 memory.

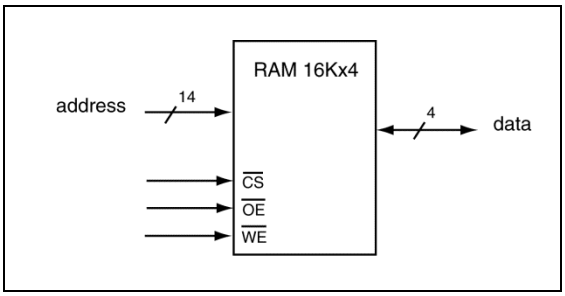
4) Show the address ranges in both binary and hexadecimal associated with the use of four 128Kx8 memories to form one 512Kx8 memory.

5) Show the address ranges in both binary and hexadecimal associated with the use of eight 1Kx8 memories to form one 8Kx8 memory.

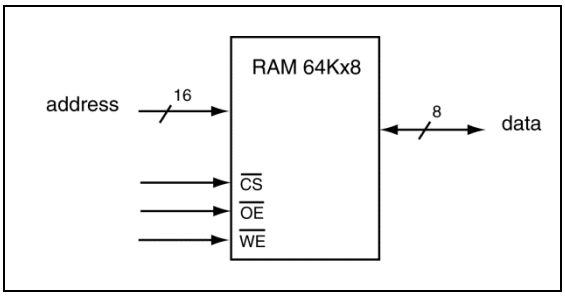
6) Show a circuit diagram that uses the following listed RAM to effectively create one memory that is 256x24. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



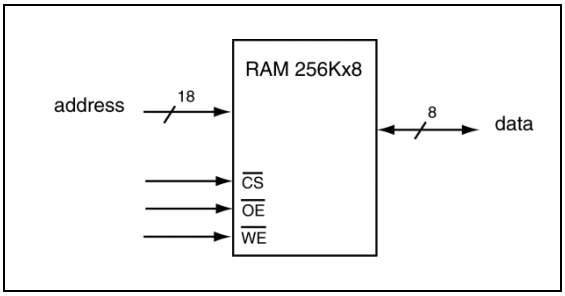
7) Show a circuit diagram that the following RAM to effectively create one memory that is 16Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



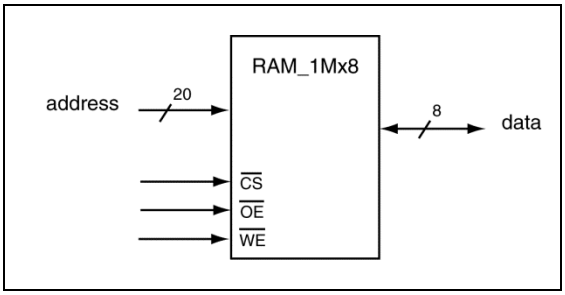
8) Show a circuit diagram that the following RAM to effectively create one memory that is 64Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



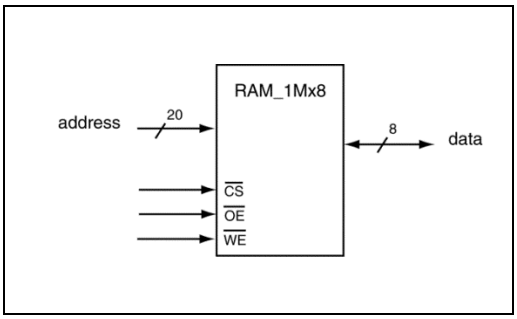
9) Show a circuit diagram that the following RAM to effectively create one memory that is 256Kx16. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



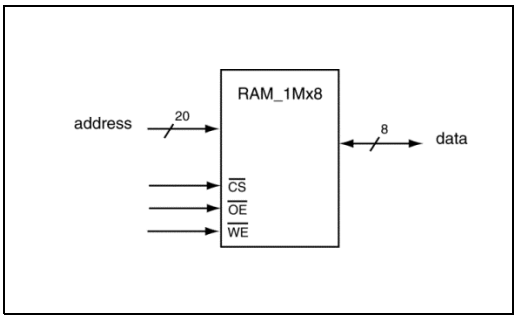
10) Show a circuit diagram that the following RAM to effectively create one memory that is 1Mx32. Assume the memory has bi-direction outputs. The control signals CS, OE, and WE are the chip selects, output enable, and write enable, respectively.



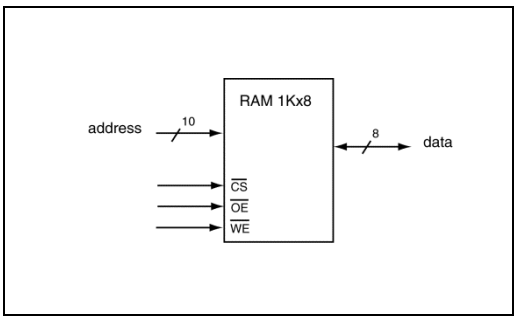
11) Use as many of the following RAMs as you need to create a memory with a 2Mx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.



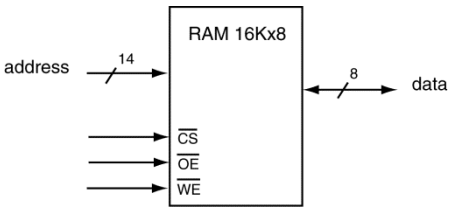
12) Use as many of the following RAMs as you need to create a memory with a 4Mx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.

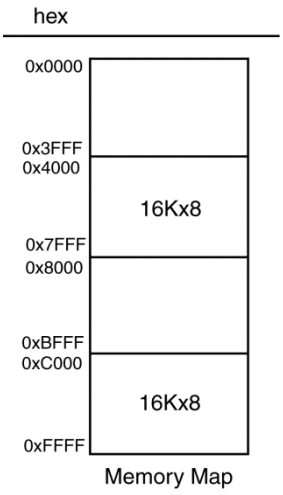


13) Use as many of the following RAMs as you need to create a memory with an 8Kx8 capacity. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively. Also, provide a memory map that shows the address space accessible by each of the underlying memories.

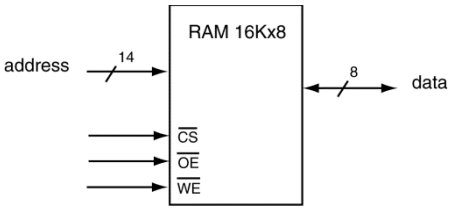


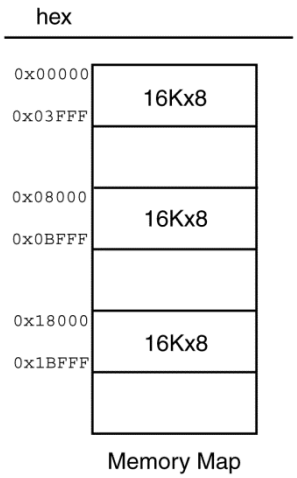
14) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only 2:4 standard decoder, and 2) using only one 1:2 standard decoder. Use two of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



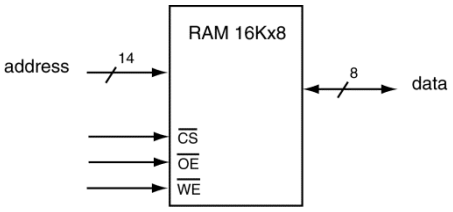


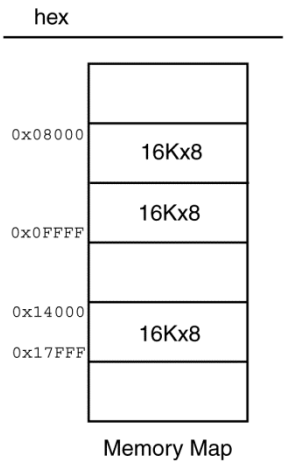
15) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only 3:8 standard decoder, and 2) using only a VHDL model for the so-called "magic_CKT". Use three of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



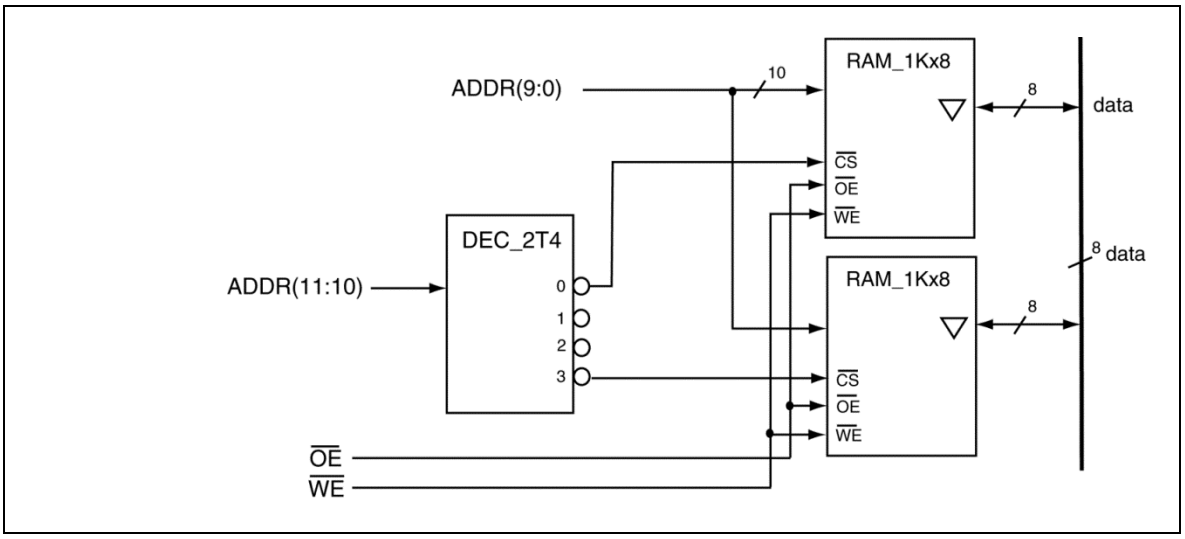


16) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only one 3:8 standard decoder, and 2) using only a VHDL model for the so-called "magic_CKT". Use three of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

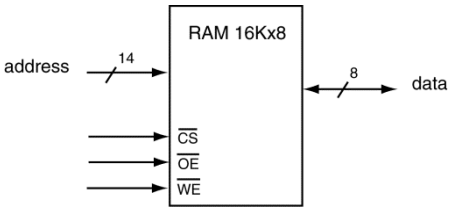


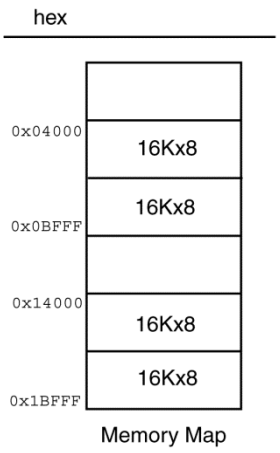


17) Complete the memory map below that describes the following design. Make sure to provide the starting and ending addresses (hex or binary) for used each memory device. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

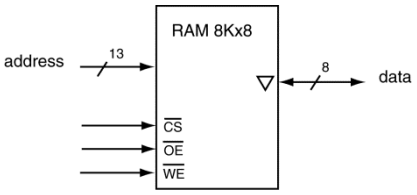


18) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only one 3:8 standard decoder, and 2) using only a VHDL model for the so-called "magic_CKT". Use four of the 16Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

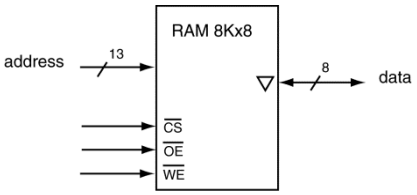


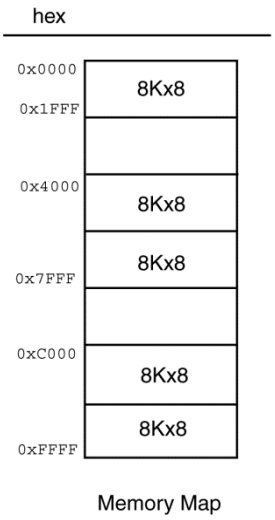


19) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only one 3:8 standard decoder, and 2) using only a VHDL model for the so-called "magic_CKT". Use four of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

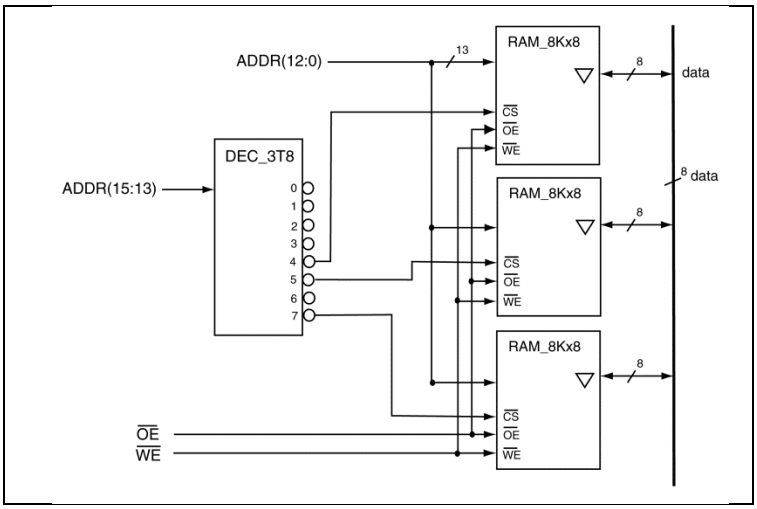


20) Design a circuit that implements the following memory map. Show two solutions to the problem, 1) Using only one 3:8 standard decoder, and 2) using only a VHDL model for the so-called "magic_CKT". Use five of the 8Kx8 RAMs listed below in your design. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.

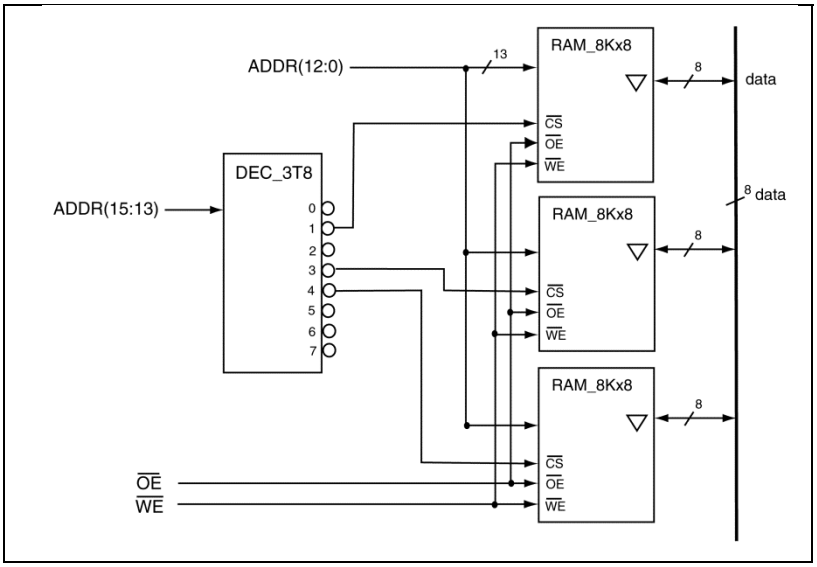




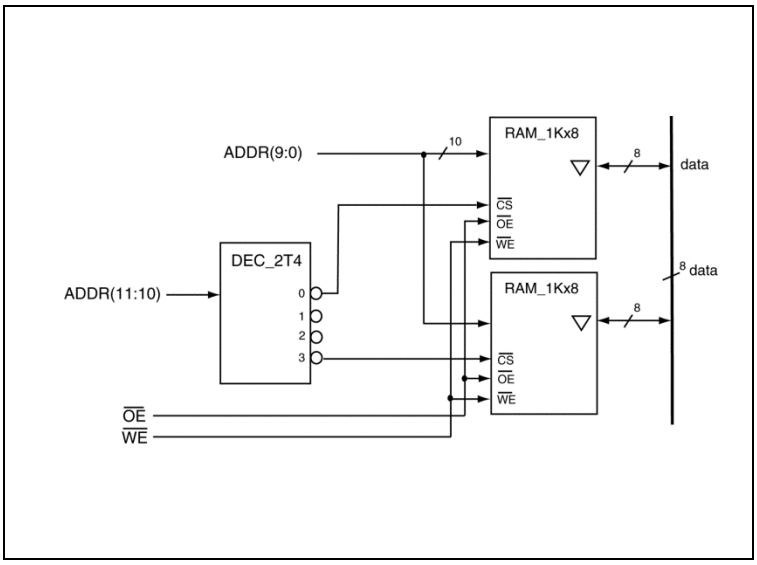
21) Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



22) Provide a memory map that you could use to describe the following design. Make sure to provide the starting and ending addresses for each memory and non-memory segment.



23) Complete the memory map below that describes the following design. Make sure to provide the starting and ending addresses (hex or binary) for used each memory device. Assume the CS input is an active-low chip select that "turns off" the device, which leaves the data outputs in a high-impedance state. The WE and OE are the write enables (writing) and output enables (reading), respectively.



24) Change this model for a generic ROM such that its capacity is 16x16. Initialize the ROM such that the data in each ROM location is eight less than its address. Draw a schematic diagram for your new model.

```
entity gen_async_rom is
    Port ( ADDR : in  STD_LOGIC_VECTOR(4 downto 0);
           DATA : out STD_LOGIC_VECTOR(3 downto 0));
end gen_async_rom;

architecture gen_async_rom of gen_async_rom is
   TYPE vect_array is ARRAY(0 to 31) of STD_LOGIC_VECTOR(3 downto
0);

   CONSTANT my_memory: vect_array := ( X"0", X"1", X"2", X"3",
                                       X"4", X"5", X"6", X"7",
                                       X"8", X"9", X"A", X"B",
                                       X"C", X"D", X"E", X"F",
                                       X"F", X"E", X"D", X"C",
                                       X"B", X"A", X"9", X"8",
                                       X"7", X"6", X"5", X"4",
                                       X"3", X"2", X"1", X"F");

begin

   DATA <= my_memory(CONV_INTEGER(ADDR));

end gen_async_rom;
```

25) Change this model for a generic ROM such that its capacity is 8x32. Model the ROM to include tri-state outputs. Initialize the ROM such that the data in the ROM is four repeated bytes of each location's associated address. Draw a schematic diagram for your new model.

```
entity gen_async_rom is
    Port ( ADDR : in  STD_LOGIC_VECTOR(4 downto 0);
           DATA : out STD_LOGIC_VECTOR(3 downto 0));
end gen_async_rom;

architecture gen_async_rom of gen_async_rom is
   TYPE vect_array is ARRAY(0 to 31) of STD_LOGIC_VECTOR(3 downto
0);

   CONSTANT my_memory: vect_array := ( X"0", X"1", X"2", X"3",
                                       X"4", X"5", X"6", X"7",
                                       X"8", X"9", X"A", X"B",
                                       X"C", X"D", X"E", X"F",
                                       X"F", X"E", X"D", X"C",
                                       X"B", X"A", X"9", X"8",
                                       X"7", X"6", X"5", X"4",
                                       X"3", X"2", X"1", X"F");

begin

   DATA <= my_memory(CONV_INTEGER(ADDR));

end gen_async_rom;
```

26) Change this model for a generic RAM such that its capacity is 16Kx16. Draw a schematic diagram for your new model.

```
entity gen_ram is
port (    CLK : in std_logic;
           WE : in std_logic;
         ADDR : in std_logic_vector(4 downto 0);
         D_IN : in std_logic_vector(3 downto 0);
        D_OUT : out std_logic_vector(3 downto 0));
end gen_ram;

architecture gen_ram of gen_ram is
   type ram_type is array (0 to 31) of std_logic_vector (3 downto
0);
   signal gen_ram : ram_type;
begin
   ram_write: process (CLK,WE)
   begin
      if (WE = '1') then
         if (rising_edge(CLK)) then
            gen_ram(conv_integer(ADDR)) <= D_IN;
         end if;
      end if;

   end process ram_write;

   D_OUT <= gen_ram(conv_integer(ADDR));

end gen_ram;
```

27) Change this model for a generic RAM such that its capacity is 64Kx32. Add the functionality to make give this RAM the ability to tri-state it's data outputs. Draw a schematic diagram for your new model.

```
entity gen_ram is
port (    CLK : in std_logic;
           WE : in std_logic;
         ADDR : in std_logic_vector(4 downto 0);
         D_IN : in std_logic_vector(3 downto 0);
        D_OUT : out std_logic_vector(3 downto 0));
end gen_ram;

architecture gen_ram of gen_ram is
   type ram_type is array (0 to 31) of std_logic_vector (3 downto
0);
   signal gen_ram : ram_type;
begin
   ram_write: process (CLK,WE)
   begin
      if (WE = '1') then
         if (rising_edge(CLK)) then
            gen_ram(conv_integer(ADDR)) <= D_IN;
         end if;
      end if;

   end process ram_write;

   D_OUT <= gen_ram(conv_integer(ADDR));

end gen_ram;
```

28) Change this model for a bi-directional RAM such that the RAM's capacity is 128Kx24. Draw a schematic diagram for your new model.

```
entity bi_dir_ram is
    Port ( BI_DIR_DATA : inout STD_LOGIC_VECTOR (3 downto 0);
           ADDR : in STD_LOGIC_VECTOR (4 downto 0);
           OE,WE,CLK : in STD_LOGIC);
end bi_dir_ram;

architecture bi_dir_ram of bi_dir_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL BD_RAM : memory := (others => (others =>'0') );
begin

   my_bi_dir: process(CLK,OE,WE,BI_DIR_DATA,ADDR,BD_RAM)
   begin
      if (OE = '1') then
         BI_DIR_DATA <= BD_RAM(conv_integer(ADDR));
      else
         BI_DIR_DATA <= (others => 'Z');

         if (WE = '1') then
            if (rising_edge(CLK)) then
                BD_RAM(conv_integer(ADDR)) <= BI_DIR_DATA;
            end if;
         end if;

      end if;
   end process my_bi_dir;

end bi_dir_ram;
```

29) Change this model for a bi-directional RAM such that the RAM's capacity is 4Kx16. Include a CS (chip select) control signal that is an active-low signal that must be asserted to allow any operations to occur on the RAM. Draw a schematic diagram for your new model.

```
entity bi_dir_ram is
    Port ( BI_DIR_DATA : inout STD_LOGIC_VECTOR (3 downto 0);
           ADDR : in STD_LOGIC_VECTOR (4 downto 0);
           OE,WE,CLK : in STD_LOGIC);
end bi_dir_ram;

architecture bi_dir_ram of bi_dir_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL BD_RAM : memory := (others => (others =>'0') );
begin

   my_bi_dir: process(CLK,OE,WE,BI_DIR_DATA,ADDR,BD_RAM)
   begin
      if (OE = '1') then
         BI_DIR_DATA <= BD_RAM(conv_integer(ADDR));
      else
         BI_DIR_DATA <= (others => 'Z');

         if (WE = '1') then
            if (rising_edge(CLK)) then
                BD_RAM(conv_integer(ADDR)) <= BI_DIR_DATA;
            end if;
         end if;

      end if;
   end process my_bi_dir;

end bi_dir_ram;
```

30) Change this model for the dual-port RAM such that it can is a 64x32 dual-port RAM. Draw a schematic diagram for your new model.

```
entity gen_dual_port_ram is
    Port ( D_IN   : in  STD_LOGIC_VECTOR (3 downto 0);
           ADRX   : in  STD_LOGIC_VECTOR (4 downto 0);
           ADRY   : in  STD_LOGIC_VECTOR (4 downto 0);
           WE     : in  STD_LOGIC;
           CLK    : in  STD_LOGIC;
           DX_OUT : out STD_LOGIC_VECTOR (3 downto 0);
           DY_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end gen_dual_port_ram;

architecture gen_dual_port_ram of gen_dual_port_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL dp_ram: memory := (others=>(others=>'0'));

begin

   process(clk,WE)
   begin
      if (WE = '1') then
         if (rising_edge(CLK)) then
            dp_ram(conv_integer(ADRX)) <= D_IN;
         end if;
      end if;
   end process;

   DX_OUT <= dp_ram(conv_integer(ADRX));
   DY_OUT <= dp_ram(conv_integer(ADRY));

end gen_dual_port_ram;
```

31) Change this model for the dual-port RAM such that it can simultaneously write to two different memory locations using different sets of data. Draw a schematic diagram of how your solution could be modeled using generic RAM devices. Draw a schematic diagram for your new model.

```
entity gen_dual_port_ram is
    Port ( D_IN   : in  STD_LOGIC_VECTOR (3 downto 0);
           ADRX   : in  STD_LOGIC_VECTOR (4 downto 0);
           ADRY   : in  STD_LOGIC_VECTOR (4 downto 0);
           WE     : in  STD_LOGIC;
           CLK    : in  STD_LOGIC;
           DX_OUT : out STD_LOGIC_VECTOR (3 downto 0);
           DY_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end gen_dual_port_ram;

architecture gen_dual_port_ram of gen_dual_port_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL dp_ram: memory := (others=>(others=>'0'));

begin

   process(clk,WE)
   begin
      if (WE = '1') then
         if (rising_edge(CLK)) then
            dp_ram(conv_integer(ADRX)) <= D_IN;
         end if;
      end if;
   end process;

   DX_OUT <= dp_ram(conv_integer(ADRX));
   DY_OUT <= dp_ram(conv_integer(ADRY));

end gen_dual_port_ram;
```

32) Change this model for the dual-port RAM such that it can write to two different memory locations using different sets of data, but not necessarily simultaneously. Draw a schematic diagram for your new model.

```
entity gen_dual_port_ram is
    Port ( D_IN   : in  STD_LOGIC_VECTOR (3 downto 0);
           ADRX   : in  STD_LOGIC_VECTOR (4 downto 0);
           ADRY   : in  STD_LOGIC_VECTOR (4 downto 0);
           WE     : in  STD_LOGIC;
           CLK    : in  STD_LOGIC;
           DX_OUT : out STD_LOGIC_VECTOR (3 downto 0);
           DY_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end gen_dual_port_ram;

architecture gen_dual_port_ram of gen_dual_port_ram is
   TYPE memory is array (0 to 31) of std_logic_vector(3 downto 0);
   SIGNAL dp_ram: memory := (others=>(others=>'0'));

begin

   process(clk,WE)
   begin
      if (WE = '1') then
         if (rising_edge(CLK)) then
            dp_ram(conv_integer(ADRX)) <= D_IN;
         end if;
      end if;
   end process;

   DX_OUT <= dp_ram(conv_integer(ADRX));
   DY_OUT <= dp_ram(conv_integer(ADRY));

end gen_dual_port_ram;
```

33) Show a block diagram, including an FSM, that you could use to solve the following problems. Also include a state diagram that describes the operation of the FSM to obtain the requested result. For these problems, do not use modules other than standard digital modules with typical inputs and outputs (counters, registers, shift register, comparators, RAMs, MUXes, RCAs, and decoders). If you use a decoder, be sure to provide an adequate model for it. Minimize the amount of hardware you use in your design including bit-widths of various modules.

a. Design a circuit that swaps the contents of two 16x8 RAM upon the receiving of a GO signal.

b. Design a circuit that adds the contents of a 16x8 RAM and stores the result in an accumulator.

c. Design a circuit that sums the odd and even contents of a 16x8 RAM. Place the odd and even sums in separate accumulators.

d. Design a circuit that counts the number values in a 16x8 RAM that are evenly divisible by 16.

e. Design a circuit that counts the number of odd and even parity values in a 16x8 RAM. Place the odd and even sums in separate accumulators.

f. Design a circuit that sorts the values in a 16x8 RAM. This circuit places the lowest value at the lowest RAM location. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

g. Design a circuit that replaces each value in a 16x8 RAM with the absolute value at that location. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

h. Design a circuit that changes the sign of each value in a 16x8 RAM. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

i. Design a circuit that counts the number of values in a 16x8 RAM that are greater than 24. Consider the contents of RAM to be unsigned binary numbers.

j. Design a circuit that counts the number of values in a 16x8 RAM that are in the range [38,78]. Consider the contents of RAM to be unsigned binary numbers.

k. Design a circuit that counts the number of values in a 16x8 RAM that are in the range [38,87]. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

l. Design a circuit that counts the number of values in a 16x8 RAM that are in the range [31,-42]. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

m. Design a circuit that reduces the magnitude of every non-zero value in a 16x8 RAM by 1. Do not change non-zero values. Consider the contents of RAM to be 8-bit signed binary numbers in RC form.

n. Design a circuit that multiples every value in a 16x8 RAM by 4. If the result exceeds 0xFF, write 0x00 in that location. Consider the contents of the RAM to be 8-bit unsigned binary numbers.

o. Provide a hardware diagram and state diagram that controls the hardware to complete the following task: Upon receiving a “GO” signal, the circuit finds the minimum value in a 16x8 RAM. Upon completion, the circuit continually outputs both the minimum value and the RAM address of that value until another GO signal is detected.

9 Arithmetic Logic Unit (ALU) Design

9.1 Introduction

No matter how you look at it, computers represent a major portion of the digital design experience. First, modern digital design uses personal computers as a major design tool. Secondly, we can argue that an underlying objective of learning digital design is to understand the notion of "computers" at many different levels. You are quickly gathering digital design skills; you're not that far away from designing a circuit that is officially a "computer". This does not mean that you'll soon be designing a "laptop PC", but a PC is not the only form of computer out there in digital-land.

This chapter presents a brief and high-level view of computers and then moves on to the describing one aspect of a computer: the Arithmetic Logic Unit, or ALU. This is another one of those subjects that people write books about and get PhDs for, so we'll not attempt to present the end-all of computer and/or ALU descriptions. What we present is an overview of ALUs constructed with low-level hardware and modeled at a high level using VHDL.

Main Chapter Topics

- **COMPUTER ARCHITECTURE OVERVIEW:** This chapter presents a brief high-level view of computer architecture as a motivation to introduce ALUs.
- **LOW-LEVEL ALU DESIGN:** This chapter describes a low-level yet limited approach to ALU design. This approach is primarily a review of modules we've already worked with.
- **HIGH-LEVEL ALU DESIGN:** This chapter describes ALU design using some of the high-level modeling techniques available in VHDL.

Why This Chapter is Important

This chapter is important because it describes several approaches to designing ALUs. This description includes an introduction to the use of variables in VHDL.

9.2 Computer Architecture Overview

Because this chapter is primarily concerned with ALU design, we'll first develop the proper context for ALUs. The ALU is a major component in computer design, so our approach in this section is to describe computers at a high-level. This will hopefully order provide an understanding of the purposes and possibilities for ALUs.

The term architecture appears often in digital-land, so often that you'll find it to have many different meanings depending on context. The best definition for "architecture" in a hardware context is that *the architecture of circuit describes the individual modules of a circuit and the connection between the modules*. Based on this definition, we can substitute the word "architecture" any time we've used the term "block diagram"[1].

[1] And often times that is what people do.

The notion of an "arithmetic logic unit" has no solid definition. Though it sounds like a circuit that contains both arithmetic and logic units, you'll find that is not always true. The modern use of the term ALU is attachable to any digital circuit since you can certainly argue that any digital circuit necessarily performs logic operations. Probably the most useful definition of an ALU is a circuit that has inputs for one or more operands, inputs for one or more controls, and output for the results. The ALU tweaks the operands as directed by the control signals in order to generate a desired result. Thus, the ALU is a box that tweaks data and generates a result; by no means is this said tweaking limited to arithmetic and logic operations.

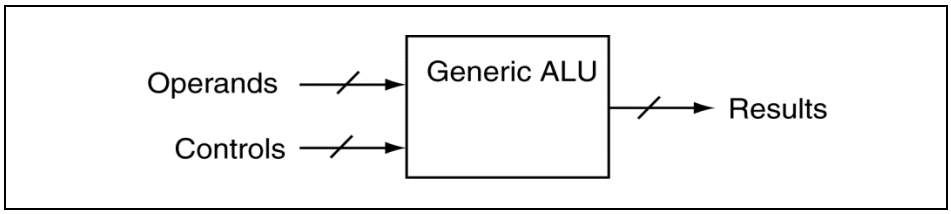


Figure 9-1: A generic block diagram of an ALU.

9.3 Computer Architecture in a Few Paragraphs

A computer is a digital system, which means that it is comprised of a bunch of gates and things that are set up in some intelligent manner such that they can do intelligent things[2]. From a higher level, a computer is nothing more than a special connection of all the standard digital circuits you've learned about up until now, plus others that you'll learn soon[3]. A computer is no different from any of the other digital systems you've worked with except that it is generally more complex. But then again, the complexity comes from the sheer amount of simple elements in the circuit and not the elements themselves[4].

What is a computer? The definition we'll work with in this discussion: A computer is any electronic device that reads instructions from memory and carries out those instructions on data. Somewhere in this definition, we need to include the notion that the computer is able to interface with the outside world, so our computer must be able to handle various input and output needs. The instructions essentially tell the computer what operations need to take place on the associated data[5].

Figure 9-2 shows the basic model of a computer that we attempted in vain to describe in the previous paragraph. As you can see, a computer is comprised of three main components: the central processing unit (CPU), memory, and input/output (I/O). The memory block stores the "instructions" that the computer executes while the I/O block allows the outside world to interface with the computer. The item we're slowly working our way to is the block labeled "CPU".

The CPU is an acronym that stands for "central processing unit" Once again, there is a lot we can say about the CPU, but we'll keep it to a comfortable minimum in this discussion. As you can see from Figure 9-2, there are two main blocks in a CPU: the "datapath" and the "control unit". The control unit interfaces with the computer instructions read from memory and tells the datapath what to do with data.

The notion of the "central" processing unit came from days where hardware was massively expensive, both on the discrete level and on the silicon level. Things are slightly different these days though. As the underlying IC fabrication techniques become better and allow for smaller digital circuitry, we rarely described the required processing in computers as "central". Having more processing units increases the overall throughput of the

[2] Such as play video games.
[3] You can survive this discussion without knowing these items.
[4] Once again, the key to understanding complex issues such as computers is to divide the associated digital circuitry into more manageable blocks. This form of abstraction is absolutely required because even the simplest computer is arguably complex. Lucky for us that VHDL structural modeling fully supports this flavor of abstraction.
[5] Instructions do other things that we're not mentioning here in order to keep things simple.

computer[6], which allows features that are more advanced on the devices in which these units appear. Where would the world be now without a device that allows you to call your buddy and watch a stream of video data all on the same device and at the same time? Now that's progress!

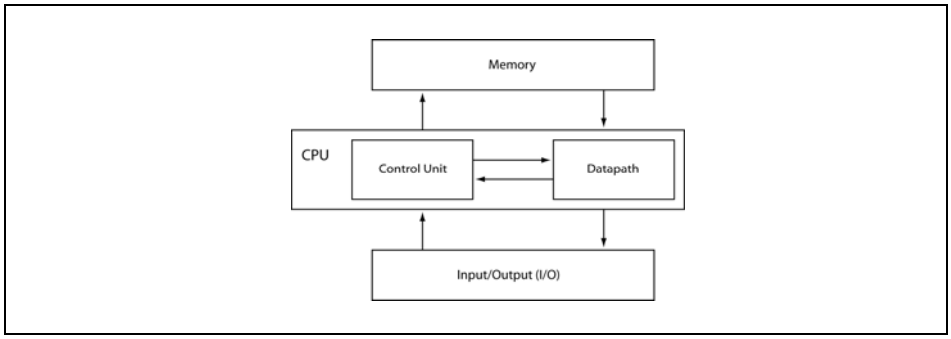


Figure 9-2: A block diagram for a basic computer architecture.

Figure 9-3 shows a more detailed diagram of the CPU. From this diagram, you can see that data passes into the datapath and then passes out of the datapath. During this datapath traversal, the hardware tweaks the data according to the instructions in instruction memory. Note that the control unit controls the datapath; this control includes receiving status from the datapath.

The datapath is the bit-crunching heart of the central processing unit (CPU). As was mentioned earlier, the datapath is a giant circuit that is filled with such a great number of simple devices that it becomes somewhat complex to study if your examination is at too low of a level. These simple devices include many types of digital circuitry, which interconnects in some intelligent and organized fashion that produces the desired result.

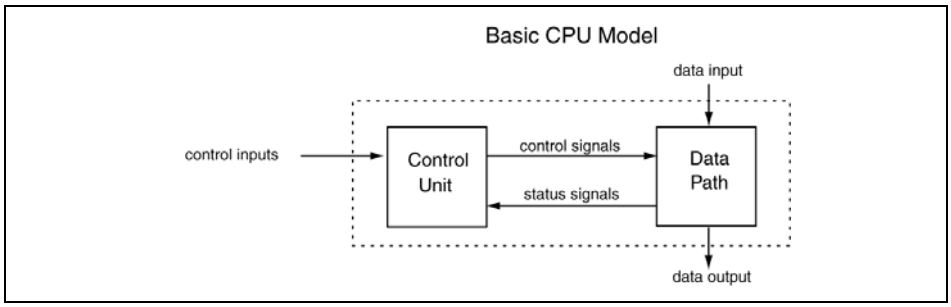


Figure 9-3: A block diagram for a basic CPU architecture.

The point we're trying to arrive at with this discussion is that the ALU forms the bulk of the datapath block and is therefore we consider it one of the basic building blocks of the CPU. Datapaths contain a buttload of useful and interesting circuitry; the only thing we'll consider here is the ALU portion of the CPU. In summary, the ALU is part of the datapath, which is part of the CPU, which is one of the three major functional blocks of a computer.

Figure 9-4 shows a lower-level diagram of a simple ALU. This is the model we'll work with in this chapter, but once again, this is by no means the only approach to ALUs out in digital-land. There are really no guidelines on how to model an ALU, which becomes truer once we start using VHDL to model ALUs.

[6] The notion here is that these various processing units are acting in "parallel", thus cranking out more junk than if you had just one processing unit.

In accordance to the acronym "ALU", Figure 9-4 shows that this particular model of an ALU contains two sub-blocks including the "arithmetic unit" and the "logic unit". In theory, all the arithmetic functions go into the arithmetic block while all of the logic functions go into the logic block. This particular ALU contains two operands: A and B[7]. The width of the operands is arbitrary as indicated by the "n" width of the operands. The S1 signal is a bundle that instructs the arithmetic unit and logic unit as to what operation to perform[8]. The width of the S1 signal depends upon what level of control is required by the two units; the more operations performed by these units, the higher value for the number "m". This model indicates that the arithmetic and logic units share the control signal; as a result, arithmetic and logic operations from these two blocks occur simultaneously. The S2 signal is another control signal that chooses between either the arithmetic or the logic result to exit the ALU.

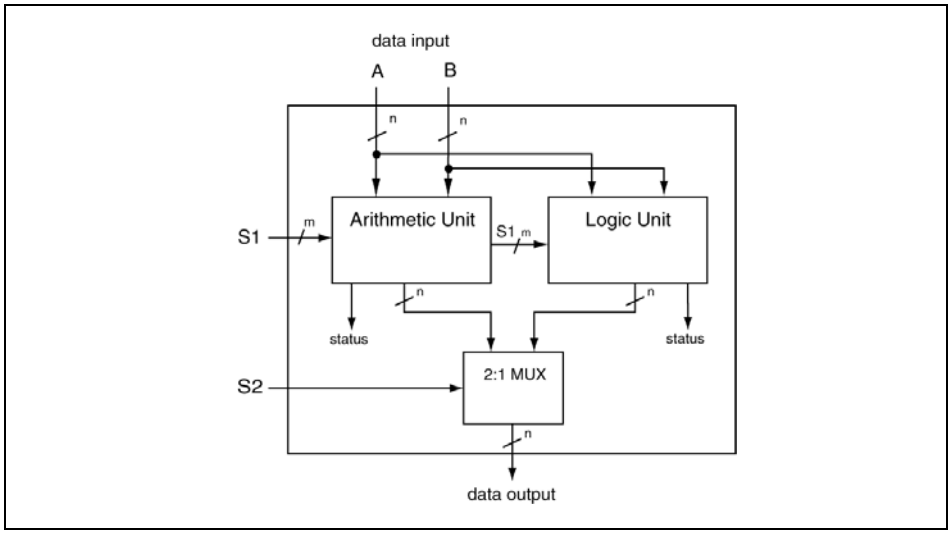


Figure 9-4: A block diagram for a basic ALU.

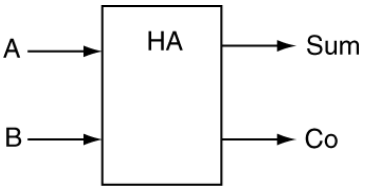
9.4 Low-Level ALU Design

You can choose one of many different ways to design ALUs. The approach we take in this section is a low-level approach that represents a review of our arithmetic-type standard digital design modules. As you will see, this low-level approach is limited, particularly when you compare it to using VHDL to model ALUs on a behavioral level. Behavioral modeling of ALUs is the topic of the next section, and I bet you can hardly wait.

9.4.1 The Arithmetic Unit

The half adder (HA) was the first circuit we designed. The HA was simple, but it gave us a view of the vast usefulness and endless possibilities of digital design. The humble HA added two 1-bit numbers; the results included a sum and carry output. Figure 9-5 shows the block diagram and the associated truth table defining the operation of the HA.

[7] ALUs can have as many operands as you feel like designing into them, though they usually have one to three operands.
[8] The associated computer instruction decides what operation needs execution.

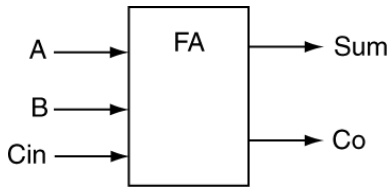


A	B	Sum	Co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) (b)

Figure 9-5: The block diagram (a) and truth table (b) for the half adder.

The HA circuit was an effective learning tool but it was not overly applicable in a real circuit. The problem with the HA was that its simplicity limited its usefulness; we could not use the circuit in a modular manner to obtain a circuit that was capable of adding more than one bit at a time. The particular limitation with the HA was the fact that it could not handle the needed input from other devices. In order to make this device useful in a modular manner, it needs to have an input that accepts and processes a carry out (Co) from another similar device. The solution to this problem leads to the notion of a full adder (FA). The FA circuit contains both a carry in (Cin) as well as a carry out (Co). Figure 9-6 shows the black box diagram and associated truth table describing the FA.



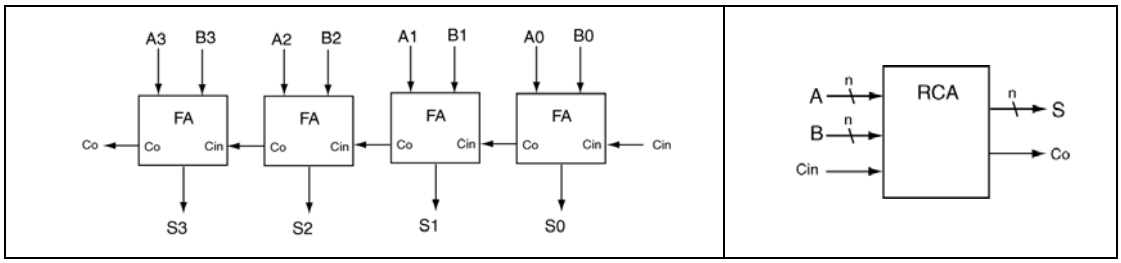
A	B	Cin	Sum	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) (b)

Figure 9-6: The full adder block diagram (a) and the associated truth table (b).

Although the FA is only capable of adding one bit at a time, you can configure many FAs to add n-bits. Applying the iterative modular design (IMD) approach moves us from a one-bit adder to an n-bit adder. The next step towards developing an arithmetic unit is to configure a set of *n* adders in such a way as to form an *n-bit* adder. This configuration requires the parallel placement of *n* FAs with the Co output of each adder driving the Cin input of the adder. In this way, the Co output of the lesser significant FA drives the Cin input of the next more significant FA in the parallel configuration.

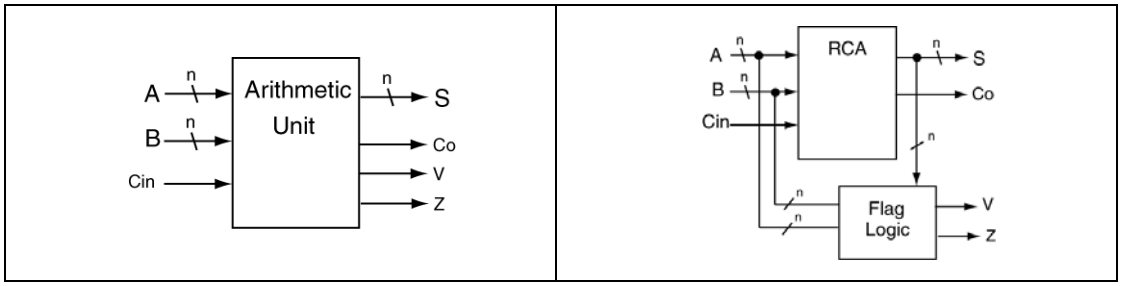
Figure 9-7 shows schematics and diagrams of a 4-bit ripple carry adder (RCA). This name comes from the notion that the carry from the less significant bits may need to transition towards the more significant bits before the output becomes valid. This feature delays the final sum output until the carry ripples through the individual FA elements in the RCA. Delayed results are undesirable in any digital circuit. However, what the RCA lacks in speed, it makes up for in simplicity.



(a) (b)

Figure 9-7: A diagram of a 4-bit adder (a) and the related n-bit ripple carry adder black box (b).

The next step in our design of an arithmetic unit is to include a few circuit additions that will be useful to us later in our discussion. Figure 9-8 shows the circuit we want to design. Note there are two extra outputs included in this circuit that were not included in the RCA circuit: V and Z. In reference to Figure 9-4, the V output indicates that an overflow condition[9] exists because of the arithmetic operation. The Z output indicates when the result of the arithmetic operation is zero. Specifically, if all the bits in the sum are zero, the Z output is set to '1'; otherwise the Z output is '0'. Figure 9-8(b) shows the approach we'll take to design the logic for the V and Z outputs; the "flag logic" will massage the augend, addend, and Sum in order to generate the V output; the Z output is a function of the S output only.



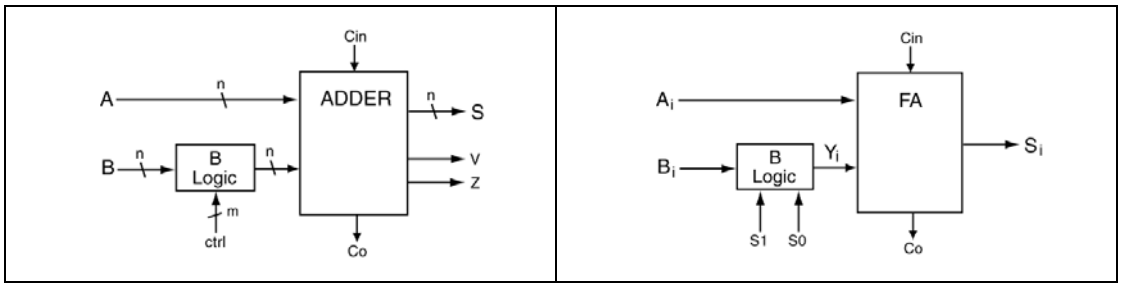
(a) (b)

Figure 9-8: The full arithmetic unit we'll design (a) and the circuit we'll use to design it (b).

Figure 9-9 shows the circuit diagrams we'll use to generate our arithmetic unit module. The approach we'll take is to massage the logic to the B input of the "adder". Note that we're referring to the box as an "adder" as we no longer model it as an RCA because of the V and Z outputs. As you'll see, if we can tweak some of the input values, we can generate several useful mathematical operations beyond the basic adding operations advertised by the RCA module.

Keeping with our digital design theme, we'll be designing the circuit shown in Figure 9-9(a) using a modular approach. Figure 9-9(b) shows the main element in this approach: the friendly FA module. Note that the FA shown in Figure 9-9(b) is drawn at the bit-level (using the subscripted i notation). We'll be designing the logic in the "B Logic" box of Figure 9-9(b). Note that there are two inputs to this box, which implies there are four different outputs for any given B input.

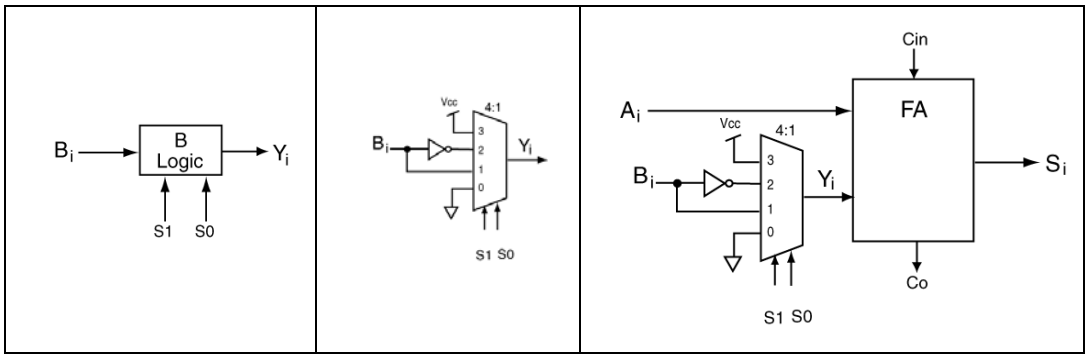
[9] In this context, an overflow condition is where the sign of the two numbers being added is the same, the sign bit of the sum is different.



(a) (b)

Figure 9-9: A block diagram of the arithmetic unit (a) and the bit-level internal element (b).

Figure 9-10(a) shows the B Logic block on the bit level. Note that this block contains three inputs (one data input and two control inputs) and one output. Yi is the arbitrary name given to the output. The two control inputs to this block "select" between one of four possible tweaks to the B input. Since the B input is only a one-bit value, the four possible tweaks to this value are 1) set the value, 2) toggle the value, 3) do nothing to the value, and 4) clear the value. Figure 9-10(b) shows the resultant MUX circuit. Placing the circuit of Figure 9-10(b) into the block diagram of Figure 9-9(b) results in Figure 9-10(c).



(a) (b) (c)

Figure 9-10: The B Logic block (a), the internal circuitry (b), and the final circuit (c).

The circuit shown in Figure 9-10(c) is a one-bit element of our final arithmetic unit. If we place "n" of these units in parallel, we would have an n-bit arithmetic circuit, which is what we've set out to do. The trick here is that by tweaking the B input, we'll be able to do more operations than the addition that is the main function of the FA element. The next task is then to see what operations our unit is capable of by listing all the possibilities in truth table format. Table 9.1 shows these possibilities.

The values in the final two columns of Table 9.1 are generated by keeping in mind that the interior of the arithmetic unit is an RCA and does nothing more than perform the operation shown in Equation 9.1. The output of the RCA (Sum or S) is the Sum of the inputs (A & B) added to the carry in (Cin). The mathematical operation shown in Equation 9.1 generates the information shown in the two right-most columns of Table 9.1. The thing to remember here is that we consider the augend and addend as signed binary numbers; Table 9.1 uses 2's compliment notation.

$$Sum = S = A + B + Cin$$

Equation 9.1: Equation for of the output of the arithmetic unit.

S1	S0	Y	Cin = 0	Cin = 1
0	0	all 0's	S = A + 0 + 0 = A (transfer)	S = A + 0 + 1 = A + 1 (increment)
0	1	B	S = A + B + 0 = A + B (addition)	S = A + B + 1 (??)
1	0	$\overline{B}$	$S = A + \overline{B} + 0 = A + \overline{B} = (??)$ (??)	$S = A + \overline{B} + 1 = (A - B)$ (subtraction)
1	1	all 1's	S = A - 1 + 0 = A - 1 (decrement)	S = A - 1 + 1 = A (transfer)

Table 9.1: The table showing possible arithmetic operations under S1 and S0 control.

The moral so far in this arithmetic design effort is that by including a chunk of logic that massages one of the operands to the RCA, the arithmetic unit went from a single operation (adding), to five valid and useful operations: addition, subtraction, increment, decrement, and a transfer[10]. Wow! You gotta love digital logic design.

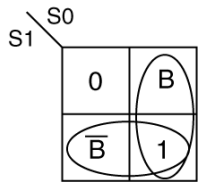
As an interesting point, we could reduce the circuitry required by the B Logic and retain the same functionality. In other words, it's possible to implement the B Logic with less circuitry than the MUX shown in Figure 9-10(c). The approach is to once again go back to your digital roots and start with a truth table in Figure 9-11(a). The Y variable represents the output of the B Logic circuit and we placed it into the compressed Karnaugh-map shown in Figure 9-11(b). Note that this K-map lists B as a mapped entered variable (MEV). Equation 9.2 shows the resulting Y logic; Figure 9-11(c) shows the final bit-level circuitry[11].

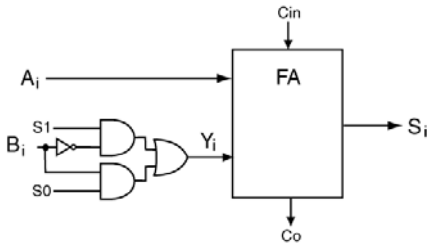
$$Y_i = S1 \cdot \overline{B_i} + S0 \cdot B_i$$

Equation 9.2: Yi output equation.

[10] You won't see it anytime soon, but the transfer action is quite useful.
[11] So, you did not read the chapter on compressed K-maps? Bummer!

S1	S0	B	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1





(a) (b) (c)

Figure 9-11: The truth table (a), compressed K-map with MEV (b), and resulting B Logic circuitry (c).

This concludes what we need to do for the arithmetic portion of this circuit. We now need to provide some logic that generates the V (overflow) and Z (zero) outputs. The best approach is to use the fact that the V output is set when the sign of the augend and addend are equal to each other but different from the sign of the result. Figure 9-12(a) shows that you can model this statement in truth table form. Note that in this truth table, the (n-1) bit position is the left-most bit (zero referenced) and hence, the sign bit for an n-bit signed binary number. Figure 9-12(b) shows the resulting equation.

$A_{(n-1)}$	$B_{(n-1)}$	$S_{(n-1)}$	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

(a)

$$V = \overline{A_{(n-1)}} \cdot \overline{B_{(n-1)}} \cdot S_{(n-1)} + A_{(n-1)} \cdot B_{(n-1)} \cdot \overline{S_{(n-1)}}$$

(b)

Figure 9-12: The truth table (a) and resulting equation for V (b).

The logic for the Z signal is more straightforward than the development of the overflow indication signal. The Z output is a '1' whenever the result of the logic operation is zero. In other words, if all the bits in the Sum are 0's, the Z output will be set; otherwise, it will be zero. This is results in a NOR operation; Figure 9-13 shows this amazing circuit.

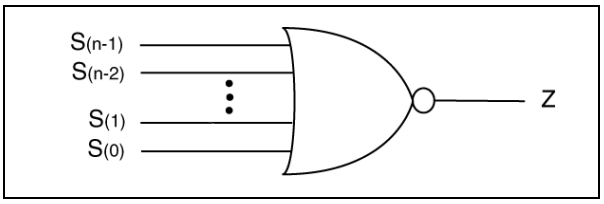
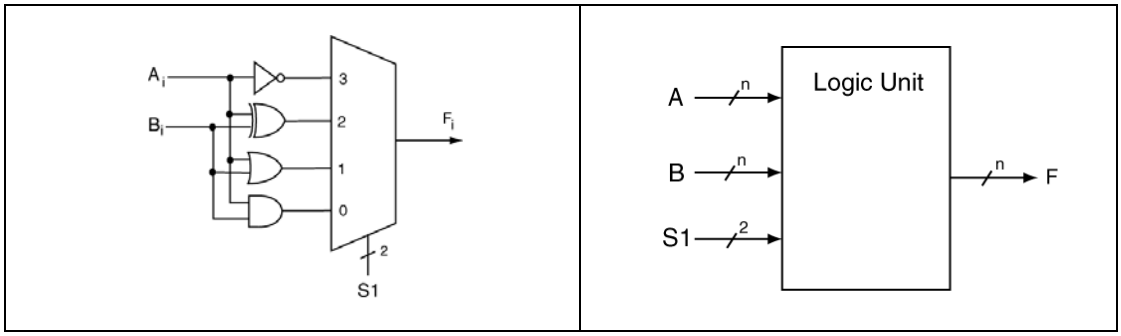


Figure 9-13: The logic for the Z output.

9.4.2 The Logic Unit

Designing the logic unit is so straightforward that we'll not need to spend much time here. What logic you choose for your ALU is up to you. For this discussion, we'll arbitrarily give our logic unit the ability to invert, XOR, OR, or AND. Specifically, the logic unit can apply these operations as follows: 1) compliment a single operand, 2) XOR two operands, 3) OR two operands, or 4) AND two operands. The logic unit chooses one of these operations based on the logic's units two control inputs. We've chosen four operations because we are reusing the control inputs that we used in the arithmetic unit (the ones that controlled the B Logic block). In our original diagram of Figure 9-4, recall that both the arithmetic and logic units share the same control inputs.

As with the arithmetic unit, we'll first show that we can model the logic unit at the bit-level. Figure 9-14(a) shows a 4:1 MUX; we consider each bit associated with the A & B operands input to one of these MUXes as part of the logic unit. The module of Figure 9-14(b) results from assembly "n" of the circuits shown in Figure 9-14(a); we refer to this as an "n-bit logic unit".



(a) (b)

Figure 9-14: The circuitry (a) and black box diagram (b) for our logic unit.

Consider for a moment that we now want to show a logic diagram for an 8-bit logic unit. We could draw eight of the circuits shown in Figure 9-14(a), but that would be a massive waste of time. A better approach would simply be model the logic unit using VHDL. Figure 9-15 shows a VHDL model for our logic unit. VHDL sure makes things much simpler.

```
entity logic_unit is
   Port (  A,B : in std_logic_vector(7 downto 0);
            S1 : in std_logic;
             F : out std_logic_vector(7 downto 0));
end logic_unit;

architecture lu of logic_unit is
begin
   with S1 select
      F <= (not A)   when "11",
           (A XOR B) when "10",
           (A OR B)  when "01",
           (A AND B) when "00",
            X"00"    when others; -- hex notation
end lu;
```

Figure 9-15: The VHDL model for the logic unit.

To complete this approach to ALU design, Figure 9-16 shows the final ALU block diagram. Note that this diagram includes three status outputs form the arithmetic unit. Any number of status inputs could have also been included from the logic unit, but we opted not to include any. One standard approach to including status outputs from both the arithmetic and logic units is to input a given status signal from each unit into a MUX. In this way, the signals selecting the individual operations on the units would also select which status output exits the ALU module.

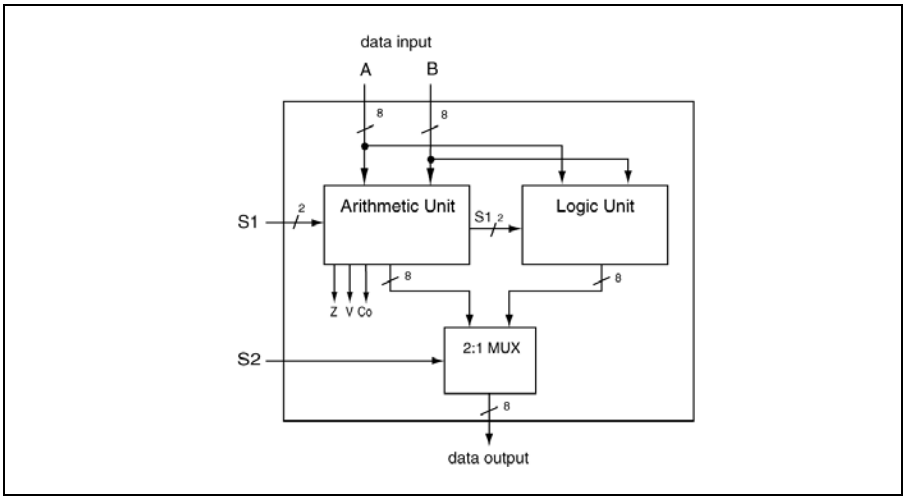


Figure 9-16: The entire ALU (8-bit form) for this section.

In the end, we now have an ALU that does quite a few operations. Table 9.2 provides a summary of the operations our ALU design can do. Here are a few things worth noting:

- S1 is a 2-bit bundle, which means there are four control inputs (S1(1), S1(0), S2, & Cin) to this ALU design. Note that we consider Cin to be a control input because it allows the ALU to generate several operations in the arithmetic unit.

- With four control inputs, we could maximally choose between 16 different operations with this ALU. Because the logic unit does not use the Cin input, the logic unit only has two control signals and thus only performs four operations (a loss of four operations for the ALU). Additionally, two control input options provide us nothing in the arithmetic side. The ALU thus performs ten operations. In truth, we have listed the "transfer A" option twice, so our ALU only performs nine unique operations.

S2	S1	Cin	Operation
0	00	-	Compliment A
0	01	-	A XOR B
0	10	-	A OR B
0	11	-	A AND B
1	00	0	Transfer A
1	00	1	A + 1 (increment A)
1	01	0	A + B (addition)
1	01	1	-
1	10	0	-
1	10	1	A – B (subtraction)
1	11	0	A – 1 (decrement A)
1	11	1	Transfer A

Table 9.2: A summary of our ALU operations.

9.5 VHDL Modeling: Signals vs. Variables

After reading through the previous section, you may find yourself hoping for a better way to design ALUs. As you may guess from the logic unit design, modeling ALUs using VHDL is straightforward. What makes this modeling straightforward is the use of a new type of VHDL object: the variable. You've been extensively using signals up to this point and hopefully found that those were no big deal. However, as you start designing more complex circuits, you'll find that using only signals in your designs is limiting. The use of variables frees you from those constraints; they also simplify the modeling of ALUs using VHDL.

Variables usage in VHDL is similar to signal usage; there are only two minor differences. We'll soon describe the two major differences as well. The minor differences lie in the notion of variable declaration and assignment. We'll describe both variable definition and assignment in the context their similarities to signals.

9.5.1 Signal vs. Variables: The Similarities

Figure 9-17 shows the similarities and differences between signal and variable declaration. Note that we declare the signals as "signal" types and we declare the variables as "variable" types; aside from that, the declarations are similar. Note that the initialization of signals and variables is the same; we can initialize them if we need to, but initialization is optional. Also, note that variables can be of subtype "std_logic" as we're used to using in the context of signals.

```
signal s_sig1 : std_logic;
signal s_sig2 : std_logic := '1';
signal s_vec1 : std_logic_vector(0 to 3);
```

(a)

```
variable v_sig1 : std_logic;
variable v_sig2 : std_logic := '1';
variable v_vec1 : std_logic_vector(0 to
3);
```

(b)

Figure 9-17: The circuitry (a) and black box diagram (b) for our logic unit.

Figure 9-18 shows the differences between signal and variable assignment. The major difference here is that assignment to signals use the signal assignment operator ("<=") while assignment to variables use the variable assignment operator (":="). Also, note that we can assign variables literal values as we've done extensively with signals. Finally, note that we can assign signals to variables and variables to signals. The notion here is that the VHDL language allows the overloading of these two assignment operators[12].

```
s_sig1 <= s_sig2;  ;
s_sig2 <= '1';
s_vec1 <= X"E";
s_sig1 <= v_sig1;
```

(a)

```
v_sig1 := v_sig2;
v_sig2 := '1';
v_vec1 := X"E";
v_sig1 := s_sig1;
```

(b)

Figure 9-18: The circuitry (a) and black box diagram (b) for our logic unit.

9.5.2 Signal vs. Variables: The Differences

There are three major differences between signal and variable usage. The first major difference is that we can only define variables in the declarative region of process while we can only declare signals in the declarative region of architectures[13]. Conversely, we can't declare variables in the declarative regions of architectures and we can't declare signals in the declarative regions of processes.

Now that we've made these statements, there are a few more issues to consider. An unmentioned similarity between signals and variables is the fact that after they retain their values after their assignment. This is an easy statement to make, but to make this more memorable, we'll soon use it in an example.

The second major difference between signals and variables is their visibility in the VHDL model. Since signals are declared in the declarative region of the architecture, a signal can be referenced anywhere in that architecture. However, since a variable declaration is part of a process statement, variables are only visible and thus usable in the process in which they are declared. This is similar to the notion of "scope" in higher-level computer programming languages[14].

The final main difference between signals and variables relates to how the VHDL synthesizer interprets them. The best way to show and described this difference is with an example. For the following example, we'll model the well-known ripple carry adder (RCA) using variables. Recall that up until now that we have only modeled RCAs using VHDL structural models. Our explanation of the differences between signals and variables should be obvious after this example.

[12] Many VHDL operators are overloaded; check the VHDL specification for full details. I actually don't know what they are and I have to check them myself when issues arise.
[13] There is actually more to the story than this. This text does not currently cover the notion of "functions" and "procedures" in VHDL. These two VHDL constructs have yet more rules for signals and variables that we'll not mention here.
[14] Once again, the notion of VHDL functions and procedures have their own rules of visibility for signals and variables. Check a VHDL reference for details.

Example 9-1: Ripple Carry Adder Behavioral Model

Model an 8-bit ripple carry adder (RCA) using VHDL variables.

Solution: Figure 9-19 shows a solution to this example. The main point of this example is that it highlights the difference between variables and signals. Honestly, if we could do this problem without the use of variables, than I admit I don't know how. The issue here is that we can easily model these problems with variables; it therefore makes no sense to find a better way to solve the problem.

The final main difference between signals and variable is that in process, the results of an assignment to a variable is available to immediately use in the process while assignments made to signals are "scheduled" to occur once the process suspends[15]. This fact has two main ramifications. First, you can use the result of a variable assignment within the process. Second, you can assign signals in processes as many time as you need to in the process; the only assignment that actually occurs is the last one seen in the process. Similarly, one way to think about signals and variables is that they store intermediate results of calculations. With variables, you can use this result immediately; with signals, the true results are not available until the process suspends.

Here are a few more worthy items to note regarding the solution shown in Figure 9-19. The number in the list below corresponds to the comments in the code.

(1) This is the variable declaration; the variable is of a std_logic_vector type of nine bits. The reason for declaring a 9-bit type with an 8-bit adder will soon become evident.

(2) This is the main addition operation. Because the variable is nine bits, we need to append an extra bit to the A & B operands; we do this with an ampersand, the operator VHDL uses for concatenation. We concatenate the left-most bit of the operand to the operand in order to ensure that operations with signed binary numbers are correct in all cases. We also add the Cin input to the "v_res" variable. Note that this statement advertises the notion that the addition operator (+) in VHDL is overloaded[16]. This is because we're adding a std_logic type to a std_logic_vector types and assigning the result to a variable. Because this is a variable assignment (note the ":=" operator), the new value is assigned to v_res immediately.

(3) At this point in the execution of the process, the v_res variable now contains the result. The SUM signal is an 8-bit signal while the v_res variable is a 9-bit signal, so we are only interested in the lower eight bits of v_res. This is signal assignment operation, so the actual assignment to the SUM signal does not occur until the process suspends. In VHDL terms, this assignment "is scheduled" to occur.

(4) As it turns out, the most significant bit of the v_res variable is the carry-out we're looking for. We assign the MSB of the v_res result to the Co signal. Because this is signal assignment, the assignment is scheduled; the actual assignment occurs when the process suspends.

[15] See the chapter on testbenches for more details.

[16] Be aware that this overloading and the liberal use of the "+" operator is based on the notion that we have included the proper libraries with our VHDL model. We always omit the library uses clauses in this text in an effort to save paper and space.

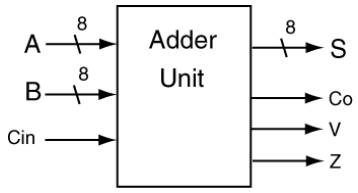
```
entity RCA_8bit is
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
end RCA_8bit;

architecture RCA_8bit of RCA_8bit is
begin
   process(A,B,Cin)
      ------------------------------------------------------(1)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      ------------------------------------------------------(2)
      v_res := ('0' & A) + ('0' & B) + Cin;

      ------------------------------------------------------(3)
      SUM <= v_res(7 downto 0);

      ------------------------------------------------------(4)
      Co <= v_res(8);

   end process;
end RCA_8bit;
```

Figure 9-19: The solution to Example 9-1.

There is one final item to note about the solution to Example 9-1. VHDL is versatile and powerful, but it lacks intelligence. As a result, you the designer need to account for many issues, particularly when arithmetic operations are involved. In the solution to this problem, the VHDL designer needs to understand that the addition of two eight-bit operands generates a nine-bit result. This is why we declared the v_res variable as nine bits. If we had not done this, we would not be able to know the value of the carry-out generated from the operation. This is true with other operations, namely multiplication.

Example 9-2: Adder Unit Behavioral Modeling

Model the following 8-bit adder unit using VHDL. Consider the S output to be a sum, the Co output to be a carryout, the Z output indicates when the S values is zero, and the V output indicates overflow from the addition of signed numbers. Consider the Cin input as a carry-in.



Solution: The first thing to notice about this problem is that it is similar to Example 9-1, which means we can use most of that solution for this problem. The main difference with this problem is that we now need to deal with an overflow (V) and zero (Z) indicator signals. The power of VHDL makes this somewhat trivial. Trivial things are good.

Here are a few more worthy items to note regarding the solution shown in Figure 9-20. The number in the list below corresponds to the comments in the code.

(1) This portion of code handles the case of the zero indicator. When the result is zero, we need the Z to be '1'; otherwise it is a '0' (can you say "positive logic"). In digital terms, we use a comparator and compare the bits forming the SUM to 0x00. As you may recall, this is easily modeled using VHDL; note that this section of code includes both an "if" and an "else" statement, which is practice in order to avoid generating unwanted latches.

(2) This piece of code handles the case of the overflow indicator. The notion here is that we define an overflow to be when the sign of the two operands are equivalent, but different from the sign of the result. This piece of code implements what the idea stated in the previous sentence. There are two items of interest here[17]. First, we initially assign the V output '0' before this piece of code is encountered. The thought here is that the code may then re-assign the signal as part of the "if" statement. While this may seem strange, you always want to make sure that you assign all of the signals you're using somewhere[18] at least once. The other thing to note here is that we use the VHDL "not equals" operator: "/=", not to be confused with a similar operator in the C programming language.

```
entity adder_unit is
   port  (  CIN : in std_logic;
          A,B : in std_logic_vector(7 downto 0);
         CO,V,Z : out std_logic;
              S : out std_logic_vector(7 downto 0));
end adder_unit;

architecture my_adder of adder_unit is
begin
   process (A,B,Cin)
      variable v_sum : std_logic_vector(8 downto 0);
   begin

      v_sum := ('0' & A) + ('0' & B) + CIN;

      CO <= v_sum(8);

      SUM <= v_sum(7 downto 0);

      ------------------------------------------------------(1)
      if (v_sum(7 downto 0) = X"00") then
         Z <= '1';
      else
         Z <= '0';
      end if;

      ------------------------------------------------------(2)
      V <= '0';
      if (A(7) = B(7)) then
         if (v_sum(7) /= A(7)) then
            V <= '1';
         end if;
      else
      end if;

   end process;
end my_adder;
```

Figure 9-20: The entire ALU (8-bit form) for this section.

[17] Please realize that there are many approaches to each section of this code; I made the subjective call that this is the clearest approach.

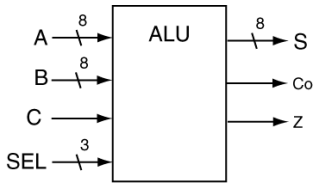
[18] Once again, the reason is that we're making sure that we always assign V is a valiant effort to avoid generating a latch.

9.6 ALU Design Using VHDL Modeling

After dragging you through this chapter, we are ready to crank out some ALU examples. The approach we'll take utilizes the full power of VHDL; we'll thus be designing ALUs on the highest level possible. While we could design both arithmetic and logic units, we'll opt to focus our designs at as high of a level of abstraction as possible.

Example 9-3: ALU Design Using Behavioral Modeling

Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out, and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



SEL	S	Co	Comment
"000"	A + B	valid	Addition
"001"	A-B	valid	Subtraction
"010"	A + B + C	valid	Addition with C
"011"	B + B	valid	2X multiplication
"100"	A + A	valid	2X multiplication
"101"	A XNOR B	-	Logic: XNOR
"110"	A NAND B	-	Logic: NAND
"111"	A AND B	-	Logic: AND

Solution: There is not much more to say about this solution that has not already been said. As you'll see from looking at the solution in Figure 9-21, we've previously described most of the VHDL usage in this model. There are a few things to note, and here they are:

(1) We treat the Z and Co outputs as signals. We give these values and overwrite the values later in the process when necessary.

(2) This is subtraction in VHDL. For this operation, we do not include the C input, which we could have used to extend the operation greater than 8-bits. The Co in this case is considered a "borrow" and has all the attributes associated with a subtraction operation.

(3) This is an addition with the C, which we consider a carry-in from another calculation. We typically use this coding style to extend the addition to something greater than 8-bits.

(4) This is the assignment of the Z indicator. Note that it was previously assigned earlier in the process; if the 8-bit result of the ALU operation is zero, this code reassigns the Z signal.

(5) This assigns the lowest eight significant bits of the intermediate result of the v_res variable to the output.

```
entity alu1 is
   port (   A,B : in std_logic_vector(7 downto 0);
              C : in std_logic;
            SEL : in std_logic_vector(2 downto 0);
          Z,Co : out std_logic;
             S : out std_logic_vector(7 downto 0));
end alu1;

architecture my_alu1 of alu1 is
begin
   process(A,B,C,SEL)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      -------------------------------------------------(1)
      Z <= '0'; Co <= '0';

      case sel is
         when "000" =>
            v_res := (‘0’ & A) + (‘0’ & B);
            Co <= v_res(8);
         when "001" =>
            -----------------------------------------(2)
            v_res := (‘1’ & A) - (‘1’ & B);
            Co <= v_res(8);
         when "010" =>
            -----------------------------------------(3)
            v_res := (‘0’ & A) + (‘0’ & B) + C;
            Co <= v_res(8);
         when "011" =>
            v_res := (‘0’ & A) + (‘0’ & A);
            Co <= v_res(8);
         when "100" =>
            v_res := (‘0’ & B) + (‘0’ & B);
            Co <= v_res(8);
         when "101" =>
            v_res := A XNOR B;
         when "110" =>
            v_res := A NAND B;
         when "111" =>
            v_res := A AND B;
         when others => v_res := (others => '1');
      end case;

      -------------------------------------------------(4)
      if (v_res(7 downto 0) = X"00") then
          Z <= '1';
      end if;

      -------------------------------------------------(5)
      S <= v_res(7 downto 0);

   end process;

end my_alu1;
```

Figure 9-21: The VHDL model solving Example 9-3.

9.7 Chapter Summary

- A computer is a device that reads instructions from memory and executes those instructions on associated data. The three main components of a computer are the memory, input/output, the central processing unit (CPU). The CPU is comprised of a control unit and a datapath; the main component of the datapath is the arithmetic logic unit (ALU).
- The term ALU can mean just about anything; ALUs are not constrained to performing only arithmetic and logic operations. ALUs can be designed on many different levels of abstraction.
- VHDL support the use of both “signals” and “variables”. Signals are declared only in the declarative regions of architectures while variables are declared only in the declarative regions of processes. Signals can be seen in all processes of an architecture while variables are only visible in the process in which they are declared. Results from signal assignment in processes are not available until the process suspends while the results from variable assignments are available immediately.
- VHDL has many overloaded operators. High-level ALU design typically utilizes the overloading of mathematical operators and the extensive use of variables in modeling of ALUs.

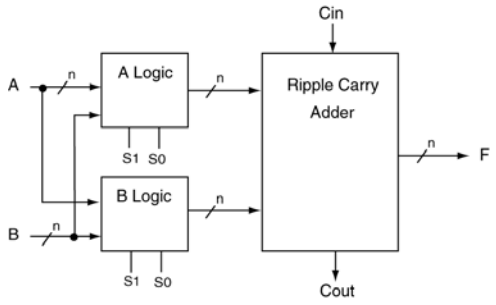
9.8 Chapter Exercises

1) Briefly describe the relationship between datapaths and ALUs.

2) Briefly describe how did "arithmetic logic units" obtain their name, as it seems ALU modules are typically capable of doing operations other than arithmetic and logic.

3) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

- Fill in the empty entries in the table below (fill in both equations and names of operations).
- Draw circuits that can be used to implement the "A Logic" and "B Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.

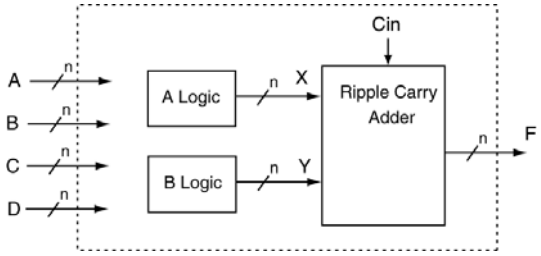


S1	S2	Cin = 0	Cin = 1
		equation	**equation**
		operation	**operation**
0	0	2A	2A + 1
			(??)
0	1	B + $\overline{B}$	
			clear (all 0's)
1	0		
		complement A	negate A
1	1	$\overline{B}$	
			negate B

4) The circuit below is a partially completed block diagram of an arithmetic circuit. The table below lists the operations required by the circuit. For this problem do the following:

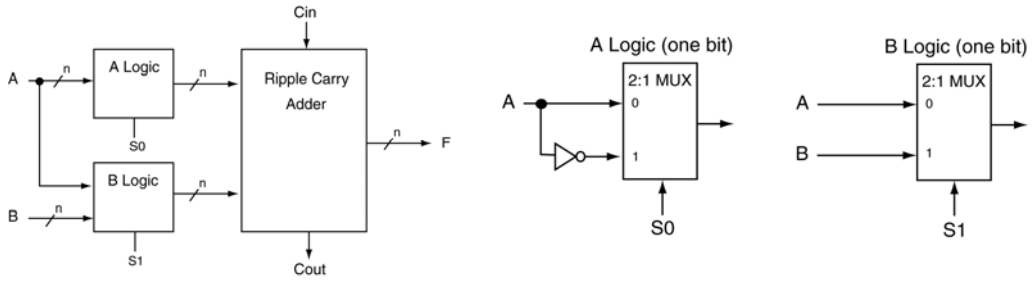
- Using only MUXes, design a bit-level implementation of the **A Logic** and **B Logic** that will implement the mathematical operations listed below. Your entire design should use no more than ***four*** control signals in addition to the **Cin** signal. Be sure to completely label your MUX control signals.

- Complete the table listed below by providing the control signals that your design uses to provide the listed operations.



Operation:	Required Control Signal Values
A - B	
A + B	
C - D	
C + D	
increment A	
decrement B	

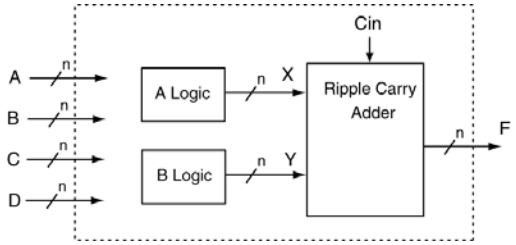
5) The figure on the left represents a block diagram of an arithmetic circuit. The diagrams on the right show the single bit versions of the "A logic" and "B logic" blocks from the diagram on the left. List all possible operations that this circuit can perform. Provide names to the arithmetic operations that make sense (such as addition, subtraction, set, clear, increment, etc.).



6) The circuit below is a partially completed block diagram of an arithmetic circuit. The table below lists the operations required by the circuit. For this problem do the following:

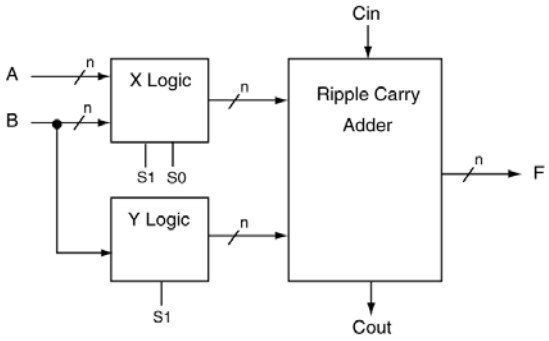
- Using only MUXes, design a bit-level implementation of the **A Logic** and **B Logic** that will implement the mathematical operations listed below. Your entire design should use no more than ***four*** control signals in addition to the **Cin** signal. Be sure to completely label your MUX control signals.

- Complete the table listed below by providing the control signals that your design uses to provide the listed operations.



Operation:	Required Control Signal Values
A - B	
A + B	
C - D	
C + D	
increment A	
decrement B	
transfer A	

7) The figure on the left represents a block diagram of an arithmetic circuit. The table on the right lists the operations that can be implemented using the arithmetic circuit. Show the logic equations or circuit implementations for a single bit for the "X Logic" and "Y Logic" blocks that implements the functionality described by the table on the right.

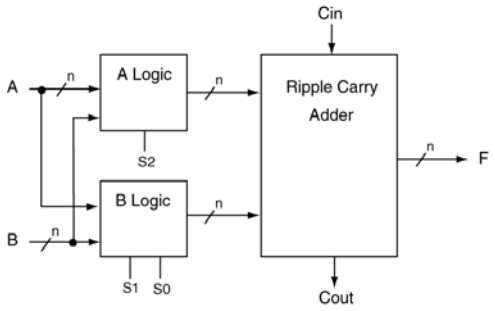


S1	S0	Cin = 0	Cin = 1
0	0	$\overline{B}+B$	0
0	1	$\overline{B}+A$	A - B
1	0	B+A	B+A+1
1	1	B-1	B

8) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

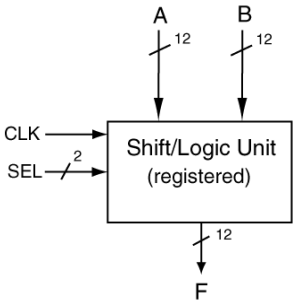
- Fill in the empty entries in the table below (fill in both equations and names of operations). *HINT: complete this bullet first; also, the two right-most columns differ by only the Cin value*.

- Draw circuits that can be used to implement the "A Logic" and "B Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.



S2	S1	S0	Cin = 0	Cin = 1
			equation	**equation**
			operation	**operation**
0	0	0	A + $\overline{B}$	
			(??)	
0	0	1	A + 0	
0	1	0		
			(decrement A)	
0	1	1		A + A + 1
				(??)
1	0	0	B + $\overline{B}$	
1	0	1		
				(increment B)
1	1	0	B - 1	
1	1	1		B + A + 1
				(??)

9) Provide a VHDL description (the architecture) of the Shift/Logic unit described below. For this problem, ignore the CLK signal. Don't use mathematical operators in your VHDL model.

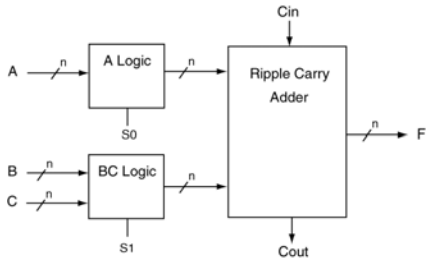


Shift/Logic Unit Specification

SEL(1)	SEL(0)	Out	Comment
0	0	$\overline{A}$	compliment A
0	1	A NOR B	bitwise NOR operation
1	0	A brl 5x	**A** barrel rotate left (5 bits)
1	1	B ÷ 16	**B** divided by 16

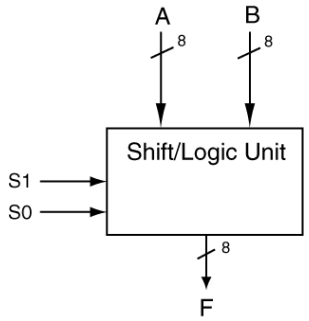
10) The circuit below is a block diagram of an arithmetic circuit. For this problem do the following:

- Fill in the empty entries in the table below (fill in both equations and names of operations).
- Draw circuits that can be used to implement the "A Logic" and "BC Logic" blocks. This circuitry should allow the arithmetic unit to implement the functionality described in the table listed below.



S1	S0	Cin = 0	Cin = 1
		equation	**equation**
		operation	**operation**
0	0		
0	1		
		(decrement B)	
1	0		A + C + 1
1	1		
		(decrement C)	

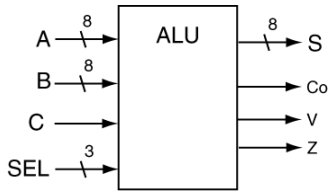
11) Provide a VHDL description of the Shift/Logic unit described below.



Shift/Logic Unit Specification

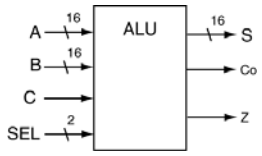
S1	S0	Out	Comment
0	0	A AND B	bitwise AND
0	1	A XOR B	bitwise exclusive OR
1	0	A asl (r-0)	A input arithmetic shift left (0 in right)
1	1	B bsr3x (l-0)	B input barrel shift right 3x (0 in left)

12) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out, the V is an overflow indicator, and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



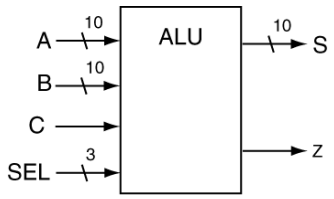
SEL	S	Co	V	Comment
"000"	A + B + C	valid	valid	Add with C
"001"	A-B-C	valid	valid	Subtract with C
"010"	B + B	valid	valid	2x B
"011"	A+ 1	valid	-	Increment A
"100"	B + 1	valid	-	Increment B
"101"	Zero	-	-	All zeros
"110"	-1	1	1	2's comp -1
"111"	A NAND B	-	-	Logic: NAND

13) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as a carry in. The S output contains the result of the selected operation. The Co is a carry-out and the Z is a zero indicator and is active for all operations. A dash in a table cell represents the case where the Co is not active and is set to '0'.



SEL	S	Co	Comment
"00"	|A| + 1	valid	Increment the absolute value of A
"01"	|B| -1	valid	Decrement the absolute value of B
"10"	|A + B|	valid	Absolute value of (A + B)
"11"	A AND B	-	Logic: A AND B

14) Provide a VHDL model for an ALU that performs the following operations. Use the SEL input to select operations; use the C input as necessary. The S output contains the result of the selected operation. The Z is a zero indicator and is active for all operations.



SEL	S	Comment
“000”	-A	Negate A
“001”	-B	Negate B
“010”	|A|	Absolute value of (A)
“011”	|B|	Absolute value of (B)
“100”	0	zero
“101”	“CCCCCCCCCC”	Propagate C input
“110”	A/2	Half A (truncate)
“111”	A OR B	Logic: AND

PART THREE: Assembly Language Programming Background and Concepts

10 Assembly Language Introduction

10.1 Introduction

Assembly language programming is overwhelming once you first see it. First, it is programming, but programming that is different from languages such as C and Python. Second, you have to learning the "instruction set" and a bunch of assembly language "tricks" to be able to program using an assembly language. Third, you probably need to become familiar with many hardware concepts regarding the computer associated with the assembly language you're setting out to learn. The bad news is there is a lot to learn, but the good news is that most all of the stuff you need to learn is relatively simple.

The problem with teaching assembly language programming is that there is no good place to start. It seems everything you need to know is based on something else you need to know, but if you're just starting out, you don't know anything. This chapter chooses to start somewhere; the stuff you learn in this chapter should help you learn the more detailed stuff in later chapters.

Main Chapter Topics

- **BEGINNERS VIEW OF ASSEMBLY LANGUAGE:** This chapter gives a generic overview of assembly languages in a context that just about anyone can understand.
- **PROGRAMMING LANGUAGE LEVELS:** This chapter put assembly language programming into a proper context of the different levels of possible for "programming a computer".
- **ASSEMBLY LANGUAGE: GOOD OR BAD:** This chapter describes some the good and bad points of using programming at the assembly language level.

Why This Chapter is Important

This chapter is important because it introduces assembly languages and associated concepts without requiring any prior knowledge of assembly languages.

10.2 Bits to Mnemonics and Back Again

We can model a computer as a device that sequentially executes a series of stored instructions. The individual instructions we use to control the various subsystems in the computer in such a way as to produce a meaningful result. When all is said and done, computers operation can be view as the pushing around of bits (1's and 0's). It should be no surprise that the computer instructions are nothing more than bits that instruct the computer to perform predefined operations.

We refer to the computer instructions in a bit-pattern form as a *machine language* or *machine code*. As you could imagine, dealing with an endless stream of bits is overwhelming for the average human brain. The solution is to replace the machine language with assembly language. An assembly language is a simple

upward translation of the machine language where the bit patterns that form the instructions are replaced by *mnemonics*. We design the assembly mnemonics in such a way as to convey the purpose of the instruction as it relates to the function that it causes the computer hardware to perform. The upside of this translation from bits to mnemonics is that the purpose of an instruction is much easier to envision and understand. The downside the bits-to-mnemonic translation is that the translation needs to be undone in order for the computer to execute the instructions.

The translation from assembly code to machine code is accomplished by a software program referred to as an *assembler*. Controlling a specific computer architecture in such a way as to do something useful requires a specific machine language, and hence, *assembly language*, for that architecture. This means there are as many different assembly languages out there as there are computer architectures. Computers generally differ by the number and type of operations, the "size" of data they work with, and the amount and methods that they store the data. From a high-level, computers are generally able to carry out essentially the same functionality, but they must do so within the limits of their underlying computer architecture. The programmer exercises the basic functionality of a computer by using the assembly language associated with a particular computer and the assembler associated with that assembly language.

10.3 Programming Language Levels

The bit patterns that make up the instructions is what controls computer: computers understand no other language. Although it is possible to write programs using the bit-patterns directly, this approach is too tedious to make is useful and there are more "useful" approaches as well. The methods used to program computers are generally broken into three general levels: 1) machine code, 2) assembly code, 3) and higher-level languages. This section describes these levels; additionally, Figure 10-1 shows a graphic of these levels.

10.3.1 Machine Code

This is the lowest level of programming. A program written in machine code is nothing more than a set of 1's and 0's. We arrange these 1's and 0's in bit-patterns that control the operations that the underlying architecture should perform. The good part about writing programs using machine code is that there is no need to use other software (not including a text editor) as a precursor to writing a program. The downside of this approach is that programs are completely unreadable, as the look like a mind-boggling stream of 1's and 0's[1]. There probably was a day when all programs had to be written in machine code, but that was sometime in the prehistoric computer era when the earth was ruled by computersaureses. Although every program that is ever written eventually ends up as machine code[2], the programs rarely start that way. Unfortunately, some instructors still require that students be able to code programs using machine code, which is nothing more than an indicator of the age and intelligence of the instructor.

10.3.2 Assembly Language

The next level up in the programming hierarchy from machine code is assembly language programming. In an assembly language, we replace the bit-patterns that form the instructions by mnemonics that loosely indicates the operation the instruction performs on the underlying computer hardware. The upside of using assembly language programming over machine code is that mnemonics bring a level of understandability to the code. The downside, (if you can consider this one) is that you need another piece of software referred to as an *assembler* to translate the assembly language instructions to machine code. The assembler is rarely a complicated piece of software. The downside of assembly language programming is that every different computer architecture (the computer hardware) will necessarily have a different assembly language. Although writing code in different assembly languages is not that complicated once you know one assembly language,

[1] You can list the machine code in hex format, but doing so does not make it more readable.
[2] More specifically, if someone executes the program on a computer.

any new assembly language will have a learning curve, with a steepness that depends on the overall complexity of the assembly language[3].

10.3.3 Higher Level Languages

The next step upwards beyond assembly language programming is to use some type of higher-level language (HLL). Because each assembly language instruction generally performs only a basic operation, assembly language programs can quickly become long (many lines of assembly instructions) when the program requires a relatively complex set of operations. One possible solution to producing long programs is switching to coding the programs using a HLL.

When you use a HLL, each line of code in the HLL can represent many lines of assembly code, which leads to shorter and arguably more understandable programs. When you use a HLL, you must use a *compiler* to translate the HLL code into machine code. Most likely, the software first converts the HLL code to assembly code before the final translation to machine code. Using a HLL has one distinct advantage over assembly code: once you know one HLL, you can write code for any computer architecture without know anything about the underlying assembly language assuming you have the correct compiler. This effectively flattens the learning curve for switching processors and makes you HLL code architecturally independent of the underlying hardware. The only downside of HLL is that the code is not necessarily as efficient as it would be if a human generated the assembly code. Compilers are good, but humans, at least some humans, are better.

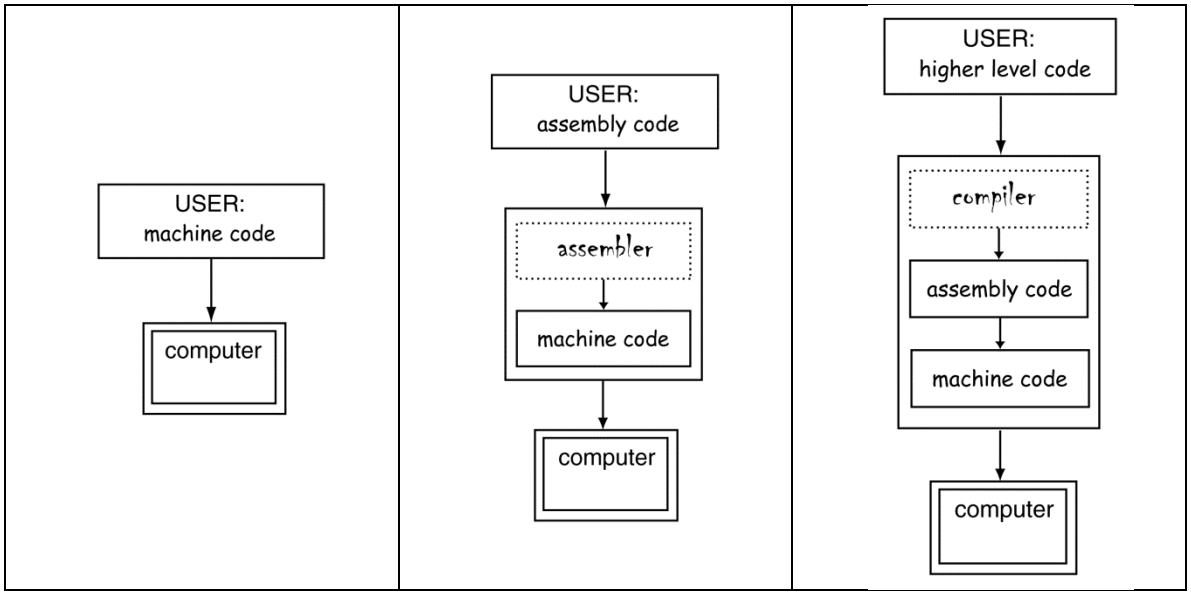


Figure 10-1: The visual choice to programming from a user's perspective.

10.4 Assembly Languages: The Low-level Goodness

Through the years, assembly languages have received some rather bad press. Most people who have worked with assembly languages find that constructing programs using assembly languages is tedious. This is because the programs tend to be "long" when they are actually performing some useful task. The length of assembly

[3] The notion here is that an assembly language instruction can be very simple or very complicated. Simple instructions, such as basic bit tweaking, are not a big deal. But, more complicated hardware can be designed to do many things with a single instruction. In this case, you're going to have to spend more time reading the manual.

programs appear to be long because we generally write the programs with only one instruction per line of text in the source code.

On the other hand, working at a low-level has several distinct advantages over using a higher-level language. Moreover, programming using a higher-level language without knowledge of the computer architecture that you intend to execute the code on can be outright inefficient in some cases. Here are some of the many benefits of programming in assembly:

- Assembly language programming inherently provides an overview of the underlying computer architecture. Therefore, writing programs using assembly language are essentially a lesson in computer programming and computer architecture all in the package.

- Assembly language programming requires that the programmer have source code organizational techniques in order to produce viable (readable, understandable, maintainable) source code. The programmer can control the potential "length" of assembly language programs by using modular programming techniques. Learning and applying these techniques will help improve the quality of your source code.

- Assembly language programming can ensure certain portions of the code operate efficiently. Even if you are primarily writing in a higher-level language, there may be portions of the code that can be "coded by hand" to make sure the machine code is as efficient as possible[4].

- There is a common argument that today's compilers are as efficient as a human (an intelligent one) programming in assembly language. If I were you, I would refuse to believe this; it sounds more like a marketing ploy for a compiler company than it does true science. Moreover, if you've ever dealt with optimizing the code generating step in a writing a compiler, you'll understand the limitations. And finally, without doubt, there are some efficient compilers out there, but it is highly unlikely that they all fall into the "good as a human" category.

- Assembly language programming helps the programmer develop a true appreciation for the higher-level languages. The more complicated the task, the more you'll want to move to a higher-level language for bulk of your programming needs. But then again, maybe not.

- Assembly language programming builds character. Yes, recent research has proven this true in practically every known case.

To be fair, Figure 10-1 alludes to one of the drawbacks of using assembly language programming. The fact is that each time you work with a different architecture, you'll need to learn a new assembly language. The upside of this downside is that learning a new assembly language, once you know at least one of them, is not that big of deal. The assorted bag of assembly language programming tricks transfers nicely between different flavors of assembly language. The way to avoid learning a new assembly language each time you switch platforms is to code in a higher-level language. The downside of this is that your code will not be as efficient and compilers are sometimes not as cost effective as their assembler counterparts are.

10.5 Assembly Languages Overview

[4] The standard trick here is to use a "profiler" to determine the typical executional characteristics of a program. As you may find out one day, programs usually spend most of their time executing a small subset of program's instructions. Therefore is you want to speed up your program without spending a lot of time doing so, you rewrite the higher-level code in these sections using assembly code. In this way, you get your program speed-up without having to recode your entire program.

There are well over 5000 different microcontrollers out there in the real world, which implies there are about the same number of different assembly languages. This number does not include the various proprietary microcontrollers and other projects that for one reason or another never were released to the general public. The question that should be asking yourself now is: "With so many assembly languages out there, what are my odds of ever using the one we're about to learn?" This is a good question. The answer is that you'll probably never see the RAT assembly language ever again after our brief foray into the device. But here's the truth: if working with RAT microcontroller represents your first experience in writing assembly code, you may initially find it challenging. If this is not your first assembly language experience, you'll find that most the knowledge and skills you developed working with previous assembly languages and architectures can be easily applied to the RAT.

What makes all assembly languages similar is that they all do the same thing: they manipulate bits. The only differences between any two assembly languages is exactly how the bits are manipulated and how the bits are stored. These two characteristics are governed by the instructions available to the programmer and the underlying hardware. To come up to speed quickly when learning a new assembly language, you simply need to understand the basic programming resources, which will allow you to use them effectively. The quickest way to do this is by perusing the *Programming Model* and the *Instruction Set*:

- Programming Model: The programming model is the programmer's view of the machine: it shows what hardware resources are available for the programmer to use. These resources include registers and other types of memory. Another useful definition for the Programming Model is the set of registers that can be manipulated by using instructions in the associated instruction set.

- Instruction Set: The instruction set lists the set of operations that the hardware can perform under control of the programmer.

One interesting point here is that there is no mention of the actual architecture of the device. There is also no mention of the external interface of the device. These are interesting points because they highlight the fact that the discussion of assembly languages generally means that we are abstracting our approach to the device to a higher level. The general thought here is that we are now going to be writing assembly language programs. We generally assume that some other fine person has implemented the device in some type of hardware setting and has setup the environment so that all we have to do is provide the working source code. Since this is not always the case, we'll be discussing some of the hardware details of the devices that are required to make your assembly language program run on the given hardware.

10.6 Chapter Summary

- An assembly language is a set of mnemonics that represent operations that the associated computer can perform. These mnemonics represents 1's and 0's, which are "assembled" by an assembler, which outputs machine code (the 1's and 0's). Assembly language programs are written using the instruction mnemonics.
- We can write computer programs at three different levels 1) machine code (low-level), 2) assembly language (medium-level), or 3) a higher-level language (high-level). No intelligent person writes programs using machine code as this approach is too tedious. Assembly language programs can become long due their relative low level compared to higher-level languages. Writing programs in higher-level languages is relatively efficient as the compiler typically generates many lines of assembly code for one line of higher-level code.
- Assembly languages are associated with specific hardware architectures. If you switch computer hardware, you necessarily need to switch assembly languages. Higher-level languages are portable in that is you switch computer hardware, you simply need to use a different compiler on the higher-level code.
- There are many good reasons why you may want to use an assembly language over a higher-level language. Writing assembly language generally allows the knowledgeable programmer to generate code that is more efficient than a typical compiler. Assembly language programming also requires the programmer to be somewhat knowledgeable about the underlying computer architecture.
- Assembly languages essentially tell the underlying hardware how exactly to push bits around. There are only so many things you can do with bits, so learning a new assembly language after you know one is relatively easy, as it mostly requires learning a new syntax and becoming familiar with the associated programmers model.

10.7 Chapter Exercises

1) Briefly describe why you can model a computer as a device that "pushes bits around".

2) Briefly describe how an assembly language program is converted into machine code.

3) Briefly describe why it is that every program ever written and executed on a computer ends up at the machine code level.

4) Briefly describe whether it would be possible to have two different assembly languages be associated with the same computer architecture.

5) Briefly describe whether it would be possible to have two different computers use the same assembly language.

6) Briefly describe why is it is that assembly language programs can quickly become long.

7) Briefly describe what an assembler is and what it does.

8) Briefly describe what a compiler is and what it does.

9) Briefly describe why assembly language programmers need to stay organized with their coding style.

10) Briefly describe why is would be important for assembly language programmers to understand the hardware associated with the computer they are writing assembly language for.

11) Briefly describe why compiler and assemblers are good at knowing there is an error in the code but much less good at figuring out the exact error.

12) Briefly describe why it is that a compiler will never be as good at optimizing code as a good and knowledgeable human.

13) Briefly describe why it is that you must learn a new assembly language if you move to a different computer architecture.

14) Briefly describe why it is that programming in a higher-level language is more portable than programming at the assembly language level.

15) Briefly surmise why it is that assemblers are "free" more often than compilers.

11 Assembly Language Instruction Set Architecture

11.1 Introduction

The heart of assembly language programming is the instruction set associated with the computer that you're planning on programming. Each assembly language instruction comprises of a set of 1's and 0's that magically somehow control the associated computer's hardware. Though the notion of the precise 1's and 0's that make up the instruction is somewhat low level, we cover them in this chapter as they provide special insights in to various aspects of computer design typically not associated with hardware. All of these issues fall under the category of "instruction set architecture", or ISA.

Main Chapter Topics

- **INSTRUCTION SET DESIGN:** This chapter covers some of the high-level details associated with designing an instruction set from scratch.
- **INSTRUCTION TYPES:** This chapter covers the various types of instructions and associated instruction formats associated with the RAT MCU including the bit assignment for the various formats.
- **ISA DESIGN ISSUES:** This chapter covers discusses a few of the important design parameters associated with ISA design.
- **MACHINE CODE DISASSEMBLY OVERVIEW:** This chapter discusses machine code disassembly in the context of the RAT's instruction set.

Why This Chapter is Important

This chapter is important because it provides a background regarding the assembly languages and particularly assembly language instruction set architectures (ISAs).

11.2 Instruction Set Design Issues

There are people out there who spend their entire lives delving into the low-level details of instruction sets and particularly, instruction set architectures (ISAs). We're not going to go too deep into the subject in this textbook, but we're going to mention some of the most basic ISA design principles. This is one of those issues where 90% of the work in ISA design goes into the final 10% of the design. What this means is that you can generate a "good" ISA without a super-significant amount of work; most of the work (the 10% part) involves squeezing as much performance out of your ISA as possible. We won't go there.

In the end, we made a decent amount of effort to make the RAT ISA efficient, but we certainly did not put out that final 10%. What we have done should give you a good idea of some of the parameters involved in ISA design. The hope here is that you realize that basic ISA design is relatively simple once you know some of the basic design parameters. Other texts on computer architecture will most certainly fill in any details we chose to

omit. Better yet, you should be able to realize that some of the RAT's ISA is something that "you could have done better[1]".

11.2.1 Instruction Set Design

There are most definitely some great theories on instruction set design out there in computerland. More likely than not, we did not apply most of these theories in the design of the RAT instruction set for reasons that aren't worth mentioning. However, what we did do was design an instruction set that contains enough instructions for the resulting computer to do something useful without having too many instructions. Along the way, we made many engineering decisions such as not including certain "helpful" instructions in order to keep the instruction set as small as possible. We'll mention these engineering design decisions as they come up throughout this text.

If you had to declare the big issues in instruction set design, you would most likely find them related to the type of computer you're designing. As best I can see it, if you're planning to design a computer, you're intending to design one of two different types of computers. We list these types below with some other details. The most important thing to realize about this instruction set design is that the instruction set we're design has nothing at all to do with the hardware that may eventually implement the instruction set. The notion here is that we're designing an instruction set to solve a particular problem; we can view the hardware design as a separate and completely independent issue that we'll need to tackle at a later time[2].

> **General-Purpose Computer**: If you're designing a general purpose computer, then you don't really know exactly how people will use the computer. It is therefore your job as the instruction set designer to provide enough instructions to do "just about anything", which means you'll be including instructions that do generic/typical operations associated with computers/computer hardware[3]. You're essentially guessing what instructions programmers and/or compiler writers will find useful; it's an educated guess, but it's still a guess.
>
> **Specific Purpose Computer**: If you're designing a specific purpose computer, you'll know exactly how people will use that computer. Designing a specific purpose computer is generally an easier task than designing a general-purpose computer because there is typically no "guessing" involved as to what the computer needs to do. In this case, you include only the instructions you know you will use, thus your computer may not be able to do everything a general purpose computer does, but it will perform you specific task better (faster, less hardware) than the general purpose computer.

11.2.2 Assembly Language Instruction Overview

Up until this point, you've probably not even seen an assembly language program, or even an assembly language instruction. This section provides an overview of what we'll delve into much deeper later. Without too much ado, Figure 11-1 shows a fragment of RAT assembly language code; the verbage that follows Figure 11-1 describes the points in that fragment that are pertinent to this discussion. Please don't worry about what exactly what the assembly code is doing (or not doing); we'll talk about that later.

- The assembly language instructions are the items in bold, capitalized italics. We usually capitalize assembly instructions, but we don't italicized or bold them; bolding and italicizing in this example highlight the instruction mnemonics.

[1] So if you find yourself thinking this, then you have really mastered the material.

[2] Another way to look at this is that the average computer science human can design an instruction set but most likely has no notion of how to design the hardware to implement those instructions. This idea here is that someone knows what they want to do but may not know how to do it in hardware.

[3] I know you probably don't have a clue to the purpose of assembly language instructions; we'll get to those details later. For now realize that assembly languages generally tweak bits in fairly standard ways, which sort of references the notion there are only so many things you can do with bits.

- There is one and only one instruction per line of text; this is a characteristic of all assembly languages.

- Some of the assembly language instruction mnemonics contain text after the instruction (and before the semi-colon). These are the "operands" associated with the individual instructions. RAT assembly language instructions have zero, one, or two operands, depending on the particular assembly instruction (much more on this later in this chapter).

- A semicolon follows each line of instruction; more text follows each semicolon. The RAT assembly language uses the semicolon to indicate a comment. Thus, the assembler ignores all the text on a given line that follows a semicolon. It is good practice in assembly language programming to comment all lines of assembly code; we'll comment more on this later.

- We delineate the assembly language program by two comments in the form of dashed lines; this makes the program happier (and we all want our programs to be happy).

```
;--------------------------------------------------------------------
init:    CLI                        ; prevent interrupts
                                    ;
main:    IN       r0,DATA_IN        ; grab data, place in r0
         EXOR     r0,0xFF           ; do something worthwhile
         OUT      r0,DATA_OUT       ; output some data
         BRN      main              ; go back to main loop
;--------------------------------------------------------------------
```

Figure 11-1: A simple, yet complete, assembly language program.

11.2.3 Instruction Formats

If you look in any other assembly language-related textbook, you'll see that they explain the instruction sets by first dividing into "types" of instructions. For reasons you'll see later, there are many gray areas regarding types of instructions. These gray areas result in a slight amount of confusion, so we'll not attempt such a division here. What we will do is divide the assembly language instructions into "format" types, as there are no gray areas there. This division will hopefully give you a quick overview and insight into the RAT instructions as well as the thought process that went into the design. We'll fill in the details in later chapters.

Figure 11-2 shows that there are five different instruction formats for all of the instructions in the RAT MCU. This means that every RAT assembly language instruction falls into one of these formats. We base the five format types on both the number and type of operands associated with the instruction. Here are some other important issues you can see in Figure 11-2; we'll provide more individual details in Section 11.3.

- Dashes ("-") in Figure 11-2 represent bits not used by the instruction. Different instruction types have varying number of unused bits.

- There can be only zero, one, or two operands for a given RAT instruction.

- There are only two types of operands: reg and immed. In this case, we do not include the lack of an operand as an operand (the none-type). In addition, there are actually two different types of immed-types.

- We encode each instruction using 18-bits, (bit-0 through bit-17 with bit-17 being the MSB), with each of the formats having different special "designated sections" within the instruction. These designated areas fall into one of two categories: *fields codes* and *operational codes*, or "op-codes".

- We use both a “G” and an “F: to designate the op-code sections for all the instructions. In other words, the combination of the G and F bits form the complete op-code for the instruction. More specifically, the G designation differentiates one of the five instruction types while the F code differentiates the particular instruction (more on this later). Four of the instructions use the exact same bits for the G and F codes, while the reg/imm-type instruction uses a slightly different set of bits for the G and F codes. There are two things about this that seem rather strange: 1) we separate the F codes in four of the instruction types, and 2) the G code is three bits wide in four of the instruction types but is only one bit in the reg/imm-type instruction. Another way to look at this is that the F codes are always four bits wide (though divided except in the reg/imm-type instruction) while the G codes are usually three bits except in the case of the reg/imm-type instruction where it is one bit.

- There are two types of field codes: 1) register fields, and 2) immediate fields.

- We use “rX” to designate a single register field for instruction types referencing only one register (reg/imm and reg-type instructions); we add a “rY” for the reg/reg-type instruction as it references two register fields. Note that each rX field uses the same bits for the instructions that have an rX operand.

- We use either “aa” or a “k” designation for the immediate fields. We use these two designations to differentiate between an 8-bit and a 10-bit immediate value, which is because these values associate with different parts of the hardware[4].

[4] This may seem cryptic now, but we’ll provide a better explanation once we discuss some of the RATs hardware details.

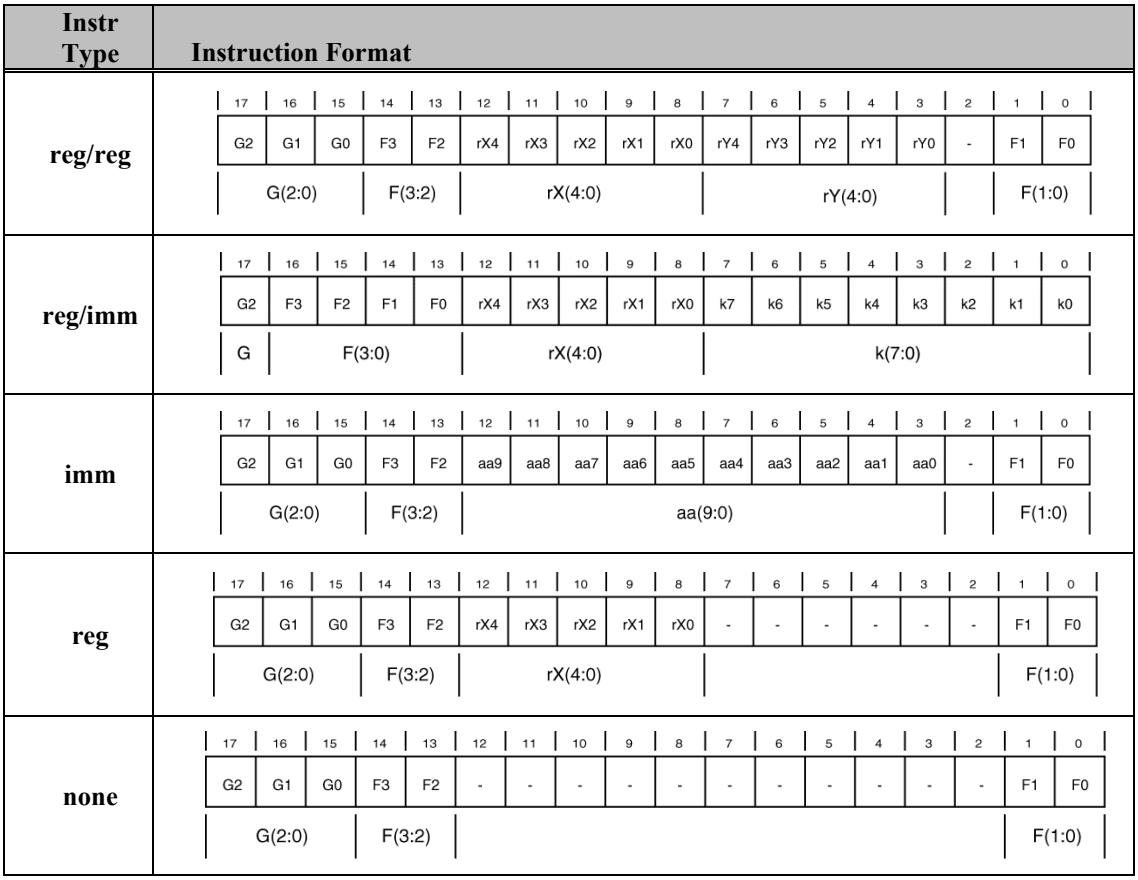


Figure 11-2: A bit-level mapping of the five different formats of RAT assembly language instructions.

11.3 RAT Instruction Types

This section describes each of the five individual instruction types associated with the RAT instruction set. The previous section mentioned these types but did not provide man interesting low-level details. Each of the following subsections provides an example instruction in the associated format, which you don't need to understand now (we'll talk about them later).

11.3.1 REG/REG-Type Instructions

Figure 11-3 shows an example of one the reg/reg-type RAT assembly language instructions. Every reg/reg-type instruction in the RAT assembly language uses this format for this type of instruction. There are other types of reg/reg instructions; this is only one of the many. The complete instruction comprises of the instruction mnemonic followed by two reg-type operands. This example arbitrarily lists an "ADD" instruction. Recall that the RAT architecture contains 32 general purpose registers (r0→r31), and you can use any combination of which for reg/reg-type instructions.

The reg/reg-type instruction comprises of op codes and field codes. As you can see from examining Figure 11-4, the opcodes are bits (17→13) and bits (1→0) while there are two field codes; one for rX (bits (12→8)) and rY (bits (7→3)). The op-code comprises of the G and F sections. The rX and rY fields reference the registers associated with (or used by) the instruction. Since there are 32 register in the RAT architecture, we must use 5-bits to differentiate the particular register operand designated by the instruction. The bit(2) contain

a dash, which indicates this bit is not used by the op-codes (to decode the instruction) or by the field codes (to differential the register). Also note that the fields are set for each instruction, meaning the same fields always represent the same bits in the instruction word.

```
ADD     r0,r4           ;example reg/reg-type instruction
```

Figure 11-3: An example assembly language instruction of the reg/reg type (with comment).

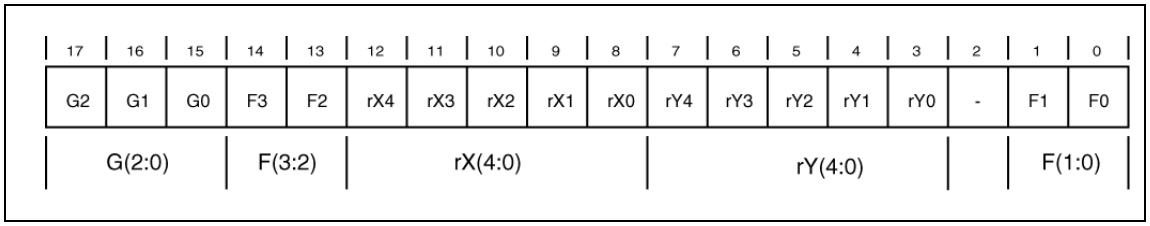


Figure 11-4: The instruction format for reg/reg-type instructions.

Error! Reference source not found. shows the entire list of reg/reg instructions including the bit assignments for each instruction. Note that we use four op-code bits to differentiate the particular reg/reg instruction. Also note that we did not use any fancy numbering for the individual instrutions; in this case, a binary code works as well as anything else. The final thing to note here is that due to the nature of the bit assignments, the RAT MCU could not possibly have more than 32 different types of reg/reg instructions. In order to obtain these 32 instructions, we would have to utilize the unused bit as in addition to F(3:0).

Instr	G(2:0)			F(3:2)		rX(4:0)					rY(4:0)					-	F(1:0)	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	0	0	0	0	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	0
OR	0	0	0	0	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	1
EXOR	0	0	0	0	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	0
TEST	0	0	0	0	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	1
ADD	0	0	0	0	1	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	0
ADDC	0	0	0	0	1	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	1
SUB	0	0	0	0	1	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	0
SUBC	0	0	0	0	1	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	1
CMP	0	0	0	1	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	0
MOV	0	0	0	1	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	0	1
LD	0	0	0	1	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	0
ST	0	0	0	1	0	rX4	rX3	rX2	rX1	rX0	rY4	rY3	rY2	rY1	rY0	-	1	1

Figure 11-5: The entire list of RAT MCU reg/reg-type instructions.

11.3.2 REG-Type Instructions

Figure 11-6 shows an example of one reg-type RAT assembly language instruction. Every reg-type instruction in the RAT assembly language uses this instruction format. There are other types of reg instructions; this is an example of one type. The complete instruction comprises of the instruction mnemonic followed by one operand. This example arbitrarily lists an "ROR" instruction, which stands for "rotate right". Since this is a reg-type instruction, the instruction necessarily has only one reg-type operand. Recall that the RAT architecture contains 32 general purpose registers (r0→r31), and you can use any one of those registers with a reg-type instruction.

The reg-type instruction comprises of op codes and field codes. As you can see from examining Figure 11-7, the opcodes are bits (17→13) and bits (1→0), and there is only one field code rX (bits (12→8)). The op-code comprises of the G and F sections. The rX field references the register associated with (or operated on) by the instruction. Since there are 32 registers in the RAT architecture, we must use 5-bits to differentiate the particular register operand designated by the instruction. The bits (7→2) contain dashes, which indicate these bits are not used by the op-codes (to decode the instruction) or by the field codes (to differentiate the register). Note that the fields are set for each instruction, meaning the same fields always represent the same bits in the instruction word.

```
ROR     r0          ;example reg-type instruction
```

Figure 11-6: An example assembly language instruction of the reg-type (with comment).

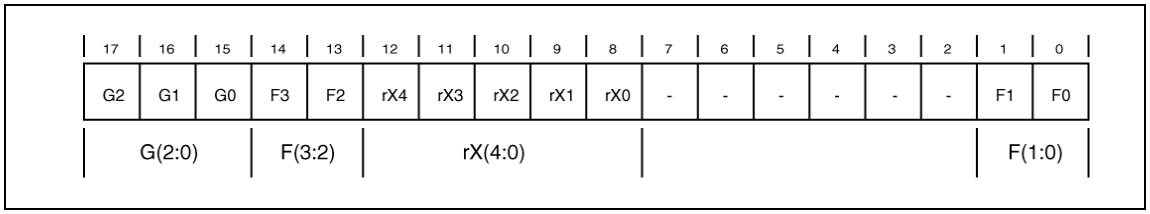


Figure 11-7: The instruction format for reg-type instructions.

Error! Reference source not found. shows the entire list of reg instructions including the bit assignments for each instruction. Note that we use four op-code bits to differentiate the particular reg instruction. Also, note that we use a binary code to differentiate the instructions.

Instr	G(2:0)			F(3:2)		rX(4:0)					-						F(1:0)	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL	0	1	0	0	0	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	0	0
LSR	0	1	0	0	0	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	0	1
ROL	0	1	0	0	0	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	1	0
ROR	0	1	0	0	0	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	1	1
ASR	0	1	0	0	1	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	0	0
PUSH	0	1	0	0	1	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	0	1
POP	0	1	0	0	1	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	1	0
WSP	0	1	0	1	0	rX4	rX3	rX2	rX1	rX0	-	-	-	-	-	-	0	0

Figure 11-8: The entire list of RAT MCU reg-type instructions.

11.3.3 REG/IMM-Type Instructions

Figure 11-9 shows an example of one the reg/imm-type RAT assembly language instruction. Every reg/imm-type instruction in the RAT assembly language uses this instruction format. There are other types of reg/imm instructions; this is an example of one of them. The complete instruction comprises of the instruction mnemonic followed by two operands. This example arbitrarily lists an "SUB" instruction, which stands for "subtraction". Since this is a reg/imm-type instruction, the instruction necessarily has one reg-type operand followed by an immed-type operand. You can use any of the RAT MCU's 32 general-purpose registers (r0→r31), a reg/imm-type instruction. This instruction uses an 8-bit immed value; there is another type of instruction that uses a 10-bit immed value.

The reg/imm-type instruction comprises of op codes and field codes. As you can see from examining Figure 11-10, the opcodes are bits (17→13) only. There are two field codes, one for register operand rX (bits (12→8)), and the other for the immed operand (7→0). The op-code comprises of the G and F sections. The rX field references the register associated with (or operated on) by the instruction, while the immed operand specifies an 8-bit value which the instruction uses directly. Since there are 32 registers in the RAT architecture, we use 5-bits to differentiate the particular register operand designated by the instruction. The bits (7→2) contain dashes, which indicate these bits are not used by the op-codes (to decode the instruction) or by the field codes (to differentiate the register). Note that the fields are set for each instruction, meaning the same fields always represent the same bits in the instruction word.

```
SUB     r0,0x45          ;example reg/imm-type instruction
```

Figure 11-9: An example assembly language instruction of the reg/imm type (with comment).

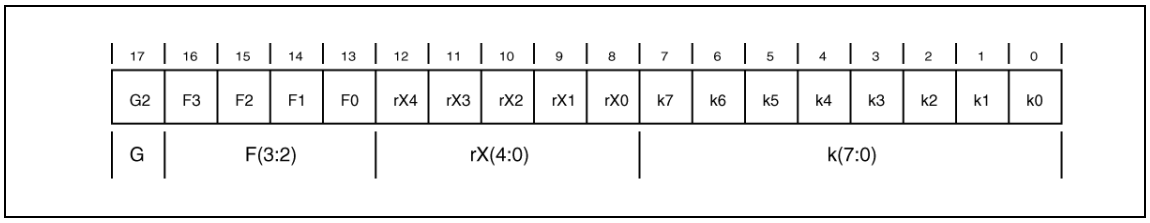


Figure 11-10: The instruction format for reg/imm-type instructions.

Error! Reference source not found. shows the entire list of reg/imm-type instructions including the bit assignments for each instruction. Note that we use four op-code bits to differentiate the particular reg instruction. Also, note that we use a binary code to differential the instructions. Lastly note that there are no unused bits, which indicates we only have instruction space for two more reg/imm-type instructions.

Instr	G	F(3:0)				rX(4:0)					k(7:0)							
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	1	0	0	0	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
OR	1	0	0	0	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
EXOR	1	0	0	1	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
TEST	1	0	0	1	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
ADD	1	0	1	0	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
ADDC	1	0	1	0	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
SUB	1	0	1	1	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
SUBC	1	0	1	1	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
CMP	1	1	0	0	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
IN	1	1	0	0	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
OUT	1	1	0	1	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
MOV	1	1	0	1	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
LD	1	1	1	0	0	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k
ST	1	1	1	0	1	rX4	rX3	rX2	rX1	rX0	k	k	k	k	k	k	k	k

Figure 11-11: The entire list of RAT reg/imm-type instructions.

11.3.4 IMMED-Type Instructions

Figure 11-12 shows an example of one immed-type RAT assembly language instruction. Every immed-type instruction in the RAT assembly language uses this instruction format. The complete instruction comprises of the instruction mnemonic followed by one operand. This example arbitrarily lists a “BRN” instruction, which stands for “branch”. Since this is an immed-type instruction, the instruction necessarily has one immed-type operand. Unlike the reg-immed instruction, the immed-type instructions use a 10-bit immed value; the reason for the difference in bit-width of the immediate values is beyond the scope of this discussion. One thing to note is that the 10-bit values are associated with the “aa” prefix (roughly meaning “address”) while the 8-bit values are associated with the “k” prefix (roughly meaning “constant”).

The immed-type instruction comprises of op codes and field codes. As you can see from examining Figure 11-13, the opcodes comprise of the G and F sections including bits (17→13) and (1→0) only. There is only one field code, which is for the immed value. The immed operand specifies a 10-bit value, which the instruction uses directly. Bit(2) contain a dash, which indicates that this bit is not used by the op-codes (to decode the instruction) or by the field codes (to differentiate the immed value). Note that the fields are set for each instruction, meaning the same fields always represent the same bits in the instruction word.

```
BRN     0x34            ;example immed-type instruction
```

Figure 11-12: An example assembly language instruction of the immed-type (with comment).

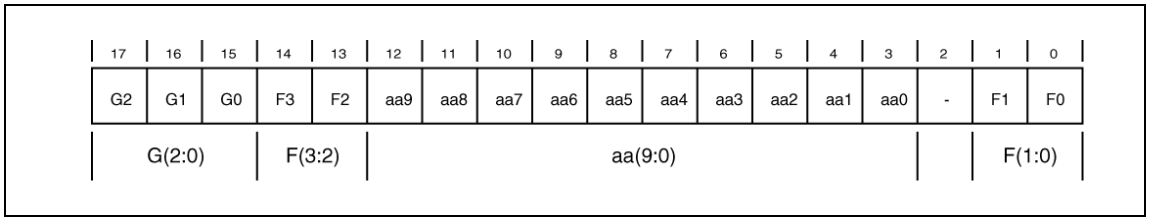


Figure 11-13: The instruction format for immed-type instructions..

Error! Reference source not found. shows the entire list of immed-type instructions including the bit assignments for each instruction. Note that we use four op-code bits to differentiate the particular immed instruction. Also, note that we use a binary code to differential the instructions.

Instr	G(2:0)			F(3:2)		aa(9:0)											F(1:0)	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRN	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	0
CALL	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	1
BREQ	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	1	0
BRNE	0	0	1	0	0	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	1	1
BRCS	0	0	1	0	1	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	0
BRCC	0	0	1	0	1	aa	aa	aa	aa	aa	aa	aa	aa	aa	aa	-	0	1

Figure 11-14: The entire list of RAT MCU reg-type instructions.

11.3.5 NONE-Type Instructions

Figure 11-15 shows an example of one none-type RAT assembly language instruction. Every none-type instruction in the RAT assembly language uses this instruction format. There are other flavors of none-type instructions; this is an arbitrary example of one instruction. The complete instruction comprises of the instruction mnemonic followed by no operands. This example arbitrarily lists a “CLI” instruction, which stands for “clear interrupt”.

The none-type instruction comprises of op codes but no field codes. As you can see from examining Figure 11-16, the opcodes are bits (17→13) and bits (1→0). The op-code comprises of the G and F sections. The bits (12→2) contain dashes, which indicate these bits are not used to decode the instruction. Note that the fields are set for each instruction, meaning the same fields always represent the same bits in the instruction word for each of the none-type instructions.

```
CLI                         ;random instruction example
```

Figure 11-15: An example assembly language instruction of the none-type (with comment).

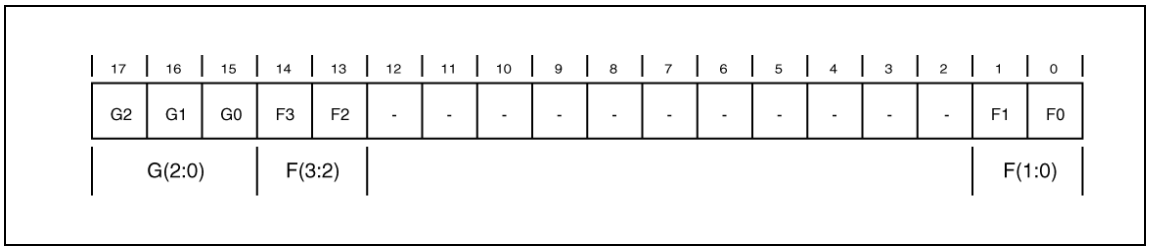


Figure 11-16: The instruction format for none-type instructions.

Error! Reference source not found. shows the entire list of none-type instructions including the bit assignments for each instruction. Note that we use four op-code bits to differentiate the particular immed instruction. Also, note that we use a binary code to differentiate the instructions.

Instr	G(2:0)			F(3:2)		-											F(1:0)	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLC	0	1	1	0	0	-	-	-	-	-	-	-	-	-	-	-	0	0
SEC	0	1	1	0	0	-	-	-	-	-	-	-	-	-	-	-	0	1
RET	0	1	1	0	0	-	-	-	-	-	-	-	-	-	-	-	1	0
SEI	0	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-	0	0
CLI	0	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-	0	1
RETID	0	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-	1	0
RETIE	0	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-	1	1

Figure 11-17: The entire list of RAT MCU none-type instructions.

11.4 ISA Design Issues

The main design considerations we took in designing the RAT's instruction formats and op-codes was to ensure all the instructions had the same bit-width and to keep the bit-width of the instructions as small as possible. You're probably wondering why the op-codes were not in the same location for all of the instruction types and why some instructions had a different number of op-code bits than other instructions. Once again, the answer is to keep the instruction bit-width as small as possible across the entire instruction set.

Here is an example of one approach we could have taken in designing the instruction formats. Since there are approximately 50 instructions in the RAT instruction set, we could have encoded those instruction op-codes using 6-bits and using the same 6-bit field in all of the instruction formats. If we had done this, we would have to look at the limiting case instruction in order to find the maximum instruction bit-width. In this case, it would be the reg/immed-type instruction. This is because the reg field requires 5-bits, the immed field requires 8 bits, and the op-code would then add six more bits. The result would be a 19-bit instruction bit-length, which is one bit more than the current 18-bits. While this does not seem like a significant number of bits, it all adds up. The RAT MCU can store 1k instructions (1024 instructions), so the extra bit would results in an extra 1k bits of required program storage. If you look at it in that sense, the extra bits do actually seem more significant.

The other painful notion here is that in most of the instruction formats, the op-code field is not contiguous in the instruction format. While this seems strange and overly complex, it's rather simple to implement in hardware. The hardware simply needs all the bits that differentiate the instruction; it does not care or does not confuse exactly from which part of the instruction the bits are routed from. You'll see this later when we discuss the RAT's control unit.

Once again, the reg/imm-type instruction is the limiting case in this approach. The reg/imm has one characteristic that the other instruction types don't have: there are no unused bits in the instruction format. You can interpret this characteristic as meaning that there is no more "instruction space" for a significant number of extra reg/imm-type instructions. As it is now, there are 14 reg/imm-type instructions and the reg/imm-type instruction format only has room for 16 instructions because there is only four bits to differentiate the various types of reg/imm instructions.

Without doubt, there are more scientific methodologies for assigning op-codes and field codes in computer instruction sets. One on hand, you could spend a lot of time and maybe shorten the instruction word-lengths by a few more bits, but it may take some effort. On the other hand, the company I used to work for did not take a scientific approach. The had approximately 70 instructions with varying field lengths; they used seven bits for the op-codes and used the worst case instruction in terms of field codes to determine the final bit length for that processor. It worked, so I can't complain too much. But then again, in terms of the RAT instructions, I wanted to strongly avoid having instructions that were 19-bits in lengths for reasons that have to do with the eventual synthesis of the RAT MCU. There is something scary and wicked about anything having an odd (non-even) bit-width out there in computer land.

11.5 Decoding the Instruction Formats

When we write programs, we generally write them in a level other than the machine code level and have some program (such as an assembler or compiler) translate the program to machine language (1's and 0's). While this is good and fine, there is also a notion of "disassembly"; this is where you take a program coded in machine code and translate it into assembly code. No sane or moral person would actually want to do this, but higher-level entities such as instructors often request if of their students.

This section gives you some insights into the disassembly process just in case some twit asks you to do this for the RAT instructions. There is really not that much to it once someone tells you how to do it; so here is what you need to know. The good part about this overview is that it provides yet more interesting insight into the genius of how the RAT assembly language instruction formats were designed. We list the steps to disassembling RAT machine code. Roughly speaking, you must first figure out the instruction type and then figure out the arguments (it any) associated with that instruction.

1) Find the bits associated with an instruction (recall that all RAT instructions are 18-bits wide)

2) Examine the MSB of the instruction (bit 17). If this bit is a '1', then you know the instruction is a reg/imm-type instruction. Otherwise, go onto step 3).

a. Examine bits (16→13) to decode which reg/imm instruction you're dealing with. Additionally, bits (12→8) specify the register number while bits (7→0) specify the immediate value.

3) Examine bits (16→15) to determine the instruction type; at this point, you know the instruction is not a reg/imm type.

a. If bits (16→15) = "00", the instruction is a reg/reg type. In this case, the bits (14, 13, 1, 0) determine which reg/reg-type instruction you're dealing with. Additionally, (12→8) determine the left register operand while bits (7→3) determine the right register operand.

b. If bits (16→15) = "01", the instruction is an immed-type. In this case, the bits (14, 13, 1, 0) determine which immed-type instruction you're dealing with. Additionally, bits (12→3) determine the 10-bit immediate value associated with the instruction.

c. If bits (16→15) = "10", the instruction is a reg type. In this case, the bits (14, 13, 1, 0) determine which reg-type instruction you're dealing with. Additionally, bits (12→8) determine the register operand.

d. If bits (16→15) = "11", the instruction is a none-type. In this case, the bits (14, 13, 1, 0) determine which none-type instruction you're dealing with.

Example 11-1: Disassembling a RAT Instruction

The following hex number represents a RAT instruction. Disassemble the instruction to determine the RAT instruction that generated this hex number: 0x18002

Solution: The first thing to note is that the MSB is '0', which tells you that this is not reg/imm-type instruction (keep in mind that you need to zero-extend the 0x18002 value). That being the case, we next examine bits (16→15), which in this case are "11", thus indicating this is a none-type instruction. From that point, we need to examine bits (14, 13, 1, 0) to find out which non-type instruction we have. Bits (14, 13, 1, 0) are "0010", which indicates the full RAT assembly instruction is "`RET`".

Example 11-2: Disassembling a RAT Instruction

The following hex number represents a RAT instruction. Disassemble the instruction to determine the RAT instruction that generated this hex number: 0x08088

Solution: The first thing to note is that the MSB is '0', which tells you that this is not reg/imm-type instruction. That being the case, we next examine bits (16→15), which in this case are "01" indicating this is an immed-type instruction. From that point, we need to examine bits (14, 13, 1, 0) to find out which non-type instruction we have. Bits (14, 13, 1, 0) are "0000" which indicate this is a `BRN` instruction. The final step is to decode the immed value, which is encoded in bits (12→3) giving us the value of 0x011. Thus, the full RAT assembly instruction is "`BRN 0x011`"

Example 11-3: Disassembling an Instruction

The following hex number represents a RAT instruction. Disassemble the instruction to determine the RAT instruction that generated this hex number: 0x27E7E

Solution: The first thing to note is that the MSB is '1', tells us that this is a reg/imm-type instruction. We then examine bits (16→13), which in this case are "0011" indicating this is a "`TEST`" instruction. Bits (12→8) provide us with the register number, which in this case is "11110", indicating r30 is the register operand for this instruction. Bits (7→0) give us the immed value, which is "0x7E". Thus the full RAT instruction in all its glory is: "`TEST r30,0x7E`".

11.6 Chapter Summary

- Instruction set architecture design can be complex or relatively straightforward depending on how much effort you put into it. If you do it correctly, you'll for sure end up with a working computer no matter how you do it.
- Computer design falls into two categories: designing general purpose computers and designing specific purpose computers.
- The act of designing an instruction sets is an independent action of designing the hardware that will be able to execute those instructions.
- Assembly languages are generally made up of an instruction mnemonic (somewhat indicating what the instruction does) followed by zero or more operands. The RAT instruction set can have zero, one, or two operands. The operands can be either registers values or immediate values. The RAT instruction set has two different types of immediate values (8-bits and 10-bits).
- Assembly language mnemonics represent carefully constructed sets of 1's and 0's, which we refer to as instruction formats. Assembly languages have different types of instructions and generally each of these instruction types have their own formats.
- Assembly language instruction formats are generally divided into opcodes and field codes. The instruction uses the op-codes to differentiate the instruction while the field codes differentiate the various operands associated with the instruction.
- There are many design issues associated with generating the instruction formats for the assembly language instruction set. The main issues are: 1) keep things simple, and, 2) keep things small. You can keep things simple by making all the instruction the same bit width and aligning as many op-code and field-codes as possible. You can keep the overall bit-width small by cleverly assigning the lengths, location, and widths of the op-codes and field codes according to your particular design needs.
- Disassembly is the art of taking raw machine code and generating the assembly language code that originally generated the machine code to begin with. Disassembly is tedious to do by hand but is quite easy to do in software or firmware. Many hardware test devices out there such as Logic Analyzers and Oscilloscopes can automatically disassemble machine code directly from the program memory outputs.

11.7 Chapter Exercises

1) In terms of instruction set design, briefly describe the two types of computer that someone may ask you to design.

2) In terms of instruction set design, briefly describe why it is "easier" to design a specific purpose computer as opposed to a general-purpose computer.

3) Briefly describe why design an instruction set is an independent function of design hardware that could implement that instruction set.

4) Briefly describe the advantage of having as many op-codes and field codes areas overlap for a given instruction set.

5) What is the greatest number of reg/reg-type instructions that the RAT MCU could have using the current instruction formats.

6) What is the greatest number of reg/imm-type instructions that the RAT MCU could have using the current instruction formats.

7) What is the greatest number of reg-type instructions that the RAT MCU could have using the current instruction formats.

8) What is the greatest number of immed-type instructions that the RAT MCU could have using the current instruction formats.

9) What is the greatest number of none-type instructions that the RAT MCU could have using the current instruction formats.

10) Which instruction type has no unused bits in its instruction format?

11) Briefly describe the main design considerations taken in designing the RAT's instruction formats and op-codes?

12) Which instruction type is the limiting case in deciding the overall bit-width of the instructions in the RAT's instruction set? Briefly describe exactly why this instruction format is the limiting case.

13) Briefly describe if it would be possible to "decompile" a program from machine code back to the original higher-level code from which it was generated.

14) Disassemble the following assembly language instructions. Clearly state the instruction type for each.

    a. `0x3A006`
    b. `0x36005`
    c. `0x1A000`
    d. `0x04302`
    e. `0x34382`
    f. `0x08199`
    g. `0x34583`
    h. `0x3650D`
    i. `0x2C701`
    j. `0x081A3`
    k. `0x18002`
    l. `0x28001`

12 Introduction to RAT Assembly Language Programming

12.1 Introduction

Assembly language programs are not complicated, but they are somewhat different from higher-level language programs you've probably written. There are many approaches you can take to learning to write assembly language programs; the approach we'll take in this chapter attempts to get you writing programs as quickly as possible. This chapter does not attempt to tell you everything you'll ever need to know about every RAT instruction in the instruction set. What we'll do instead is arbitrarily tell you only what you need to know to enable you to write and understand basic assembly language program. We'll be adding more information in later chapters as you gain more RAT assembly language programming prowess.

There are many ways of writing assembly language source code. As with all languages, you can make the source code museum quality or you can make it so no human can possibly understand it. The approach we'll take in this chapter is to outline what it takes to generate neat and organized assembly code by presenting time-tested rules and source coding techniques. It's up to you whether you choose to use them or not.

Main Chapter Topics

- **ASSEMBLY LANGUAGE PROGRAM STRUCTURE:** This chapter outlines the basic and preferred structure of assembly language programs including comments, assembler directives, and assembly language source code.
- **INTRODUCTION TO EMBEDDED SYSTEMS:** This chapter presents the notion of an embedded system as it relates to basic assembly language programming.
- **RAT I/O INSTRUCTIONS:** This chapter describes the basic instruction-level I/O characteristics of the RAT MCU.
- **RAT EXAMPLE PROGRAMS:** This chapter provides several basic RAT assembly language example programs with full explanations.

Why This Chapter is Important

This chapter is important because it describes the basic structure of assembly language programs and provides several well-commented assembly language example programs.

12.2 RAT Assembly Language Program Structure

There are three basic parts to any assembly language program: 1) comments, 2) assembler directives, and 3) the assembly language source code. The only thing you need to make your program run is assembly code, but the other parts of your program are important to having a useful and meaningful programming experience. This correctly implies that the comments and assembler directives are generally there to increase the readability, reusability, and understandability of the assembly code. You should make an effort to use them in this way.

12.2.1 RAT Assembly Language Program Comments

Without doubt, you've written computer programs before. If your source code was of high quality, then it also was necessarily well-commented. The exclusive purpose of comments is to transfer information from the source code text to other humans who may be reading your source code. Keep in mind, there is nothing uglier than a program that contains no comments[1]. A working piece of code is almost worthless without adequate comments.

Commenting source code is somewhat of an art form. There are specific rules you can read about, but the overall general rule for writing comments is that is makes your code easier for a human to understand. You of course can under-comment your source code, but you can also over comment. Writing comments for assembly language programs is somewhat different from writing comments for higher-level languages. Here are a few rules regarding comments in assembly language source code. The best way to learn how to comment assembly language code is to follow the "better" examples you'll find in this text[2]. Keep in mind that the current RAT assembler only allows single-line comments and does not support block comments.

- The RAT assembler uses a semicolon for the comment character. Thus, the RAT assembler ignores all the text following the semicolon on the line in which the semicolon appears.
- The RAT assembler only supports single-line comments; the RAT assembler does not support block-style comments common in higher-level languages such as C.
- The first set of comments that should appear in an assembly language program is the title banner. This set of comments should contain information such as names of the programmer, revision history, and a brief but adequate description of the source code in the given file.
- Multi-line comments should delineate and describe pertinent blocks of source code, such as subroutines, initialization code, complicated algorithms, etc.
- Comments should describe what the assembly code is doing in human terms, not in terms of the associated instruction mnemonics.
- Each line of assembly code (each assembly language instruction) should contain a comment unless it is patently obvious what the code is doing. Comments should provide useful information and should not repeat what is obvious from the instruction mnemonic and operands.
- Blank lines are not official comments (they're officially "whitespace") but you should use them as you would a comment to increase the readability of your source code.

12.2.2 RAT Assembler Directives

Assembler directives provide a means of communication between the programmer and the assembler. Assembler directives are similar to preprocessor directives found in higher-level computer languages such as C. Assembler directives are different from instructions, so be careful not to confuse the two different items.

Any assembler you work with will have a different set of assembler directives. There are many directives associated with the RAT assembler; we'll discuss most of the assembler directives in the context of the assembly language programming examples that follow in this and other chapters. There is a chapter dedicated

[1] Although not commenting your code provides for some measure of job protection, as does writing crappy source code.
[2] Keep in mind that some examples are not well-commented due to the fact that many source code examples are fragments of code and not complete programs.

to the RAT assembler; this chapter provides a description of all the assembler directives. We're opting not to introduce the assembler directives in this section, as they will make more sense when we introduce them in the context of RAT programming examples. Additionally, the RAT Assembler Manual lists and describes all of the available RAT assembler directives.

12.2.3 RAT Assembly Source Code

The assembly code provides a means for the programmer to communicate with the underlying hardware that forms the RAT MCU. The assembly source code consists of RAT assembly instructions that the programmer arranges in such a way as to do something useful. The appearance of your RAT assembly source code is important; examples of good-looking source code appear in the examples found in this text. Additionally, there is a RAT assembly "style-file" found in the appendix of this text as well as in the RAT Assembler Manual. This file will help you create assembly language programs that are nothing short of beautiful. If you follow the guidelines this style-file provides, your code will be readable, easy to understand, and evoke worldwide joy and celebration. You'll also be able to easily reuse your code for the various RAT-based assembly programs. A significant portion of the remainder of this text deals with the RAT assembly language instructions and their proper usage in RAT assembly language program. That being the case, we don't have too much to say about it here.

12.3 Embedded Systems Programming

As we get closer to talking about actual programming, let's describe the ultimate goal. Most assembly language programs end up in some embedded system. An embedded system is a computer-based system that requires no outside user intervention in order for it to run properly. This means the system fires up into a working state and stays working for as long as the system remains powered. Note that this is different from what you may be familiar with in your higher-level language programming courses. The programs you wrote in those courses typically did something relatively useful, and then "ended". The notion of most embedded systems is that the program they are running never ends, unless of course you remove power from the circuit. The reality of embedded systems is that they just sit there waiting to react to inputs or conditions from the outside world.

The notion of programs that never end is what you'll see in every RAT program. This is the model we'll start out with in our first RAT assembly program as well as all the programs that follow. This may take some getting used to, but it's not that big of a deal once you work with it for a while. The truth is that the RAT programming set has no instructions that stop the RAT hardware from executing instructions. Some processors out there do have such instructions, but we have no need for such an instruction in the RAT instruction set.

12.3.1 Software vs. Firmware

Often times when you're generating source code, a question of semantics often arises. When you are writing code, are you writing "software" or are you writing "firmware". Regardless of the particular hardware you're writing the code for, some portion of memory in the computer you're programming is dedicated to the storage of your program. The instructions that make up your program tell the computer exactly how to process data and what to do with the data it processes. If the user can change program memory, we consider the program stored in memory as *software* and we refer to the computer system a *general purpose* system. If the user cannot change the program in program memory, we consider the program as *firmware*, as it was written for a *single purpose* computer[3].

[3] The reality is that most people use the term *software* in reference to true software or true firmware. In most cases, this is OK because you know what the person using this term intended because of the context it was used in. The term *firmware*, on the other hand, is never used to mean software. The biggest mistake that people generally make is that they think that firmware

Many people mistakenly conclude that they are writing firmware if they are simply writing their source code using an assembly language. This is not correct. According to our definitions of firmware and software, you can write firmware using either assembly language or a higher-level language. Likewise, you can also write software using either assembly language or a higher-level language.

12.4 Input/Output (I/O)

The block diagram shown in Figure 12-1 represents another view of a basic computer system. The microprocessor is able to communicate with the various functional blocks in a computer system. We are mainly interested in the ways the MCU communicates with the outside word. Keep in mind that the only reason that computers are useful is because they are able to communicate with the outside world. It's true that computers crunch data really fast, but this speed would be useless if it were not able to transfer data and results to and from a real environment. Communications with the outside world occur through the I/O block; Figure 12-1 shows a model of this communication. The MCU is responsible for crunching data and the memory is responsible for storing the program and intermediate results. We can characterize the communication with the outside world as one of three types: 1) Programmed I/O, 2) Interrupt Driven I/O, and 3) Direct Memory Access (DMA); below is a short description of these ideas.

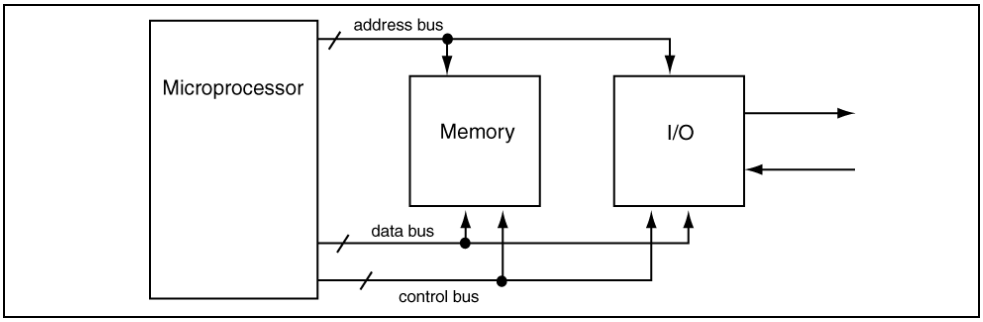


Figure 12-1: The basic computer model at a lower level.

1) Programmed I/O: Programmed I/O falls into one of two main categories: Port mapped and memory mapped. These two categories are similar from a programmer's perspective; their main differences lie in the underlying hardware implementation. We refer to this approach as "programmed" I/O because true input or output happens as a result of issuing an instruction, which the underlying program issues. The main thing to keep in mind about performing I/O is that the MCU is getting (input) or giving (output) something; therefore the program must both state what it is you're getting or giving and which external devices you're getting it from (input) or giving it to (output). In this way, an output-type instruction provides a source of data from the microprocessor to output to the destination associated with a given port_id. Similarly, an input instruction provides a destination within the microprocessor, which receives data from an external source associated with a given port_id. The differences between port mapped and memory mapped I/O lies in how exactly you specify the external device you're performing the I/O with (the source for inputting and the destination for outputting. The next two bullets describe those differences in more detail[4].

    a) **Port Mapped I/O:** Port mapped I/O uses a "port number", or "port ID", or "port_id" to specify the external device associated with the given I/O instruction. The port_id is just a number; roughly speaking, the external hardware uses (or, "decodes" maybe be a better word) the port_id

has a direct connection to assembly language programming. The reality is that firmware can be in the form of assembly language or a higher-level language (or both). Don't fall into this trap.

[4] Keep in mind this is a deep subject; you should consult other references for even more details.

to "activate" a given I/O device. In this way, every I/O device will necessarily have a unique port_id number. The port_ids are a function of the hardware; if you're writing assembly code for a given piece of hardware, someone must tell you, the programmer, the specific port_ids for the given I/O devices. We consider architectures that use port mapping as having *separate address space* for I/O, which may seem strange, but will hopefully make sense when after you read about memory mapped I/O. And finally, typical port mapped architectures have dedicated instructions for I/O, such as IN and OUT instructions for input and output, respectively.

b) **Memory Mapped I/O:** In contrast to port mapped I/O, memory mapped I/O does not have dedicated IN-type and OUT-type instructions. The memory mapped approach uses memory access instructions to handle I/O. As you may guess, memory access instructions must provide an address in memory of the item you're trying to write or store (output) or trying to read or load (input). In a memory mapped architecture, the hardware designer configured the hardware such that if you read or write from a "special address" associated with an I/O device, you're not really reading or writing to that device; you're actually input data from or outputting data to that particular device, respectively. Each I/O devices has its own unique address similar to port mapped I/O. In memory mapped systems, we consider the I/O to be sharing the address space with the data memory (recall that port mapped systems have a separate I/O space). Once again, the hardware designer must provide you the programmer with the address values associated with various I/O devices; you would not know the addresses otherwise.

There are some special issues associated with programmed I/O. In the context of output, your MCU will know when it needs to output data to an external device. The problem arises when dealing with input. Many peripheral devices require that you "get data from them" when they're ready to give it to you. The problem is that you generally do not know when such devices are ready to give you data. The only solution is to constantly ask these devices if they're ready to give you data; we refer to this process as *polling*. The reality is that if your processor is stuck polling something, it means it is not available to do other things. Another way to look at this is that it is a waste of processing power. Wasting processor power is actually not a big deal unless there is some other important task that needs doing while you're polling.

2) Interrupt I/O: Instead of the processor constantly asking if an input device is ready to provide data, or an output device is ready to receive data, it's better (in terms or processing efficiency) to have the devices tell the processor when they're ready to act. Having devices communicate directly with the processor happens via the interrupt mechanism on a given MCU. When a device is ready to communicate with the MCU, we generally refer to this as the external device is "requesting service" from the MCU. We refer to this mechanism as an "interrupt" because the processor stops what it is currently doing (stops executing the code it is currently executing) in order to take care of the device requesting service. This mechanism actually switches processing to a different set of instructions when the MCU receives an interrupt. When the external device, or "peripheral"[5] is satisfied, the MCU returns to the code it was executing before it received the interrupt.

3) Direct Memory Access (DMA): The final type of I/O is another form of I/O that does not require an excessive amount of processing power from the processor. This type of I/O is generally associated with large data transfers between memory and peripherals (as you may have gathered from the name). The idea here is that the processor limits its involvement with transfers. The concept of DMA is relatively simple but is more complex in cases where you need to actually design it or program it.

We can characterize the three types of I/O by what device is in control of handling I/O. For programmed I/O, the MCU is in charge. With interrupt driven I/O, some external device is in control. With DMA, some device

[5] Devices in digital systems are often referred to as *peripherals*. This is nothing more than saying that there is a module there that is communicating in some way with the processor.

external to the MCU is also in control. Which device is in control of the MCU's resources is a hot issue in computerland.

One of the three major subsystems of a computer is the Input/Output module. This makes sense because for programs to do anything useful, they'll need to have contact with the outside world. The general model is that the RAT will input (read) some data from the outside world, process this data, and then output (write) some result back to the outside world. There are several mechanisms that microcontrollers typically use to handle I/O; the RAT MCU uses what we refer to as *port-mapped I/O*. With the RAT, there are specific instructions used to handle input and output, which officially makes the RAT a *port mapped* device[6] . Not surprisingly, the instructions which handle input and output are the "IN" and "OUT" instructions.

The RAT MCU handles the transfer of data between the RAT core and the outside world via "ports", which is effectively a resource sharing utility. For now, you can treat ports as registers, though there is more to them than that. The notion of a port is actually tough to describe on a firmware level; when we discuss the RAT hardware, the notion of ports become much clearer. There's really not much to them, but we'll stick to their basic usage in the IN and OUT instructions for now.

The RAT hardware has the ability to have 256 unique input ports and 256 unique output ports. Programmers reference the ports by numbers; we represent each port with an 8-bit value. We reference this 8-bit number as the *port_id*. We *"address" s*pecific input and output devices external to the RAT MCU using the port_id. In that the port_id is 8-bits, up to 2^8 (256) different devices can be accessed for both input and output. Each different input and output device will necessarily have a unique port address, which we assume has been previously configured in hardware. Someone needs to tell you the programmer what the port_ids are for a given piece of hardware. Once again, we'll talk about those concepts later; for now, let's introduce the input and output instructions.

12.4.1 The IN and OUT Instructions

The IN and OUT instructions give the RAT MCU the ability to input data from the outside world (the IN instruction) or output data to the outside world (the OUT instruction). Both of these instructions share similar characteristics in form and function. The IN instruction inputs data from the outside world and puts that information in a register, while the OUT instruction outputs data from a register to the outside worlds. Recall that all RAT registers are 8-bits wide, thus all I/O associated with the RAT MCU is necessarily 8-bits wide also.

Table 12.1 shows both the IN and OUT instructions. As will all instructions in the RAT instruction set, we can best describe the functionality of the instructions using an RTL statement. Table 12.1 describes both the IN and OUT instructions using RTL statements and example usage of both instructions. Most importantly here, you should read the RAT Assembly Manual description of these instructions for the full story; the verbage we provide here is a short version. We only present the short version here because all of this will make more sense once you use the instruction in an actual assembly language program.

Instruction	RTL	Example
IN	Rd ← in_port(immed_val)	`IN Rd,imm_val`
OUT	out_port(immed_val) ← Rd	`OUT Rd,imm_val`

Table 12.1: Examples of the RAT MCU I/O instructions: IN and OUT.

[6] The I/O for any given processor can be specified as either *port mapped* or *memory mapped*. In memory mapped devices, I/O is treated exactly as a memory access (reading and writing). We'll discuss this in a later chapter.

There are important things to note about Table 12.1. First, the "immed_val" listed in both the RTL statements and in the examples, represent the port_ids associated with the instruction. As previously noted, the port_ids are 8-bit values. They appear in the instructions as "immed_val", so it is not explicitly evident what exactly they represent. The second thing to notice is that the two instructions have the same form in terms of assembly languages, but they have different forms in terms of the underlying RTL statements. Note that the RTL statements are appear somewhat opposite of each other. These differences turn up often in assembly languages, which creates sort of an issue. You would like there to be 100% consistency in the form of the assembly language, but that level of consistency is often missing. In the case of these instructions, the lack of consistency sometimes tricks you into making syntax errors when you're writing source code. There is truly no way around this; the only solution is that you must be familiar with the assembly language and not be afraid to reference the full text description of the instruction in the RAT Assembler Manual.

12.5 Introductory RAT MCU Example Programs

The example programs that follow will hopefully give you a basic understanding of RAT assembly language programs as well as a few of the directive options available in the assembler. This introduction is somewhat quick; we'll be going a bit slower and including more detail in sections. Most of the examples that follow are missing title banners in an effort to save space, but know that you should always use them in every program you write.

Example 12.1: The First RAT MCU Program

Write a basic RAT assembly language program. It does not have to do anything meaningful.

Solution: Example 12.1 shows the solution to this open-ended example; some notes and other useful information follows this solution.

```
(00)
(01)  ;-----------------------------------------------------------------
(02)  ;- Programmer: Pat Smith
(03)  ;-
(04)  ;- Description: This program is an example RAT assembly language
(05)  ;-  program; it does very little as it is our first program.
(06)  ;-----------------------------------------------------------------
(07)  ;
(08)  ;-----------------------------------------------------------------
(09)  ;- Assembler Directives
(10)  ;-----------------------------------------------------------------
(11)  .CSEG          ; indicates code segment
(12)  .ORG   0x00    ; sets the code segment counter to 0x00
(13)  ;-----------------------------------------------------------------
(14)
(15)  ;-----------------------------------------------------------------
(16)  ;- Initializations
(17)  ;-----------------------------------------------------------------
(18)            CLI                  ; prevent interrupts
(19)  ;-----------------------------------------------------------------
(20)
(21)  ;-----------------------------------------------------------------
(22)  main:     OR       r0,0x00     ; do nothing (nop)
(23)            BRN      main        ; go back to main loop
      ;-----------------------------------------------------------------
```

Figure 12-2: Solution to Example 12.1.

- Lines (00-05) provide an informative file banner. Every good piece of source code will include such an item. You really can't put too much information in the file banner. One helpful thing is a revision history in the file banner. Include one if you can.

- Lines (07-12) provide a section where you can place as many assembler directives as possible. The assembler directive is a message from the programmer to the assembler. This is yet another approach to making your source code understandable and inviting to the human reader.

- Line (10) shows a ".CSEG" is an assembler directive while line (11) shows an ".ORG" assembler directive. The "CSEG" directive tells the assembler that everything that follows will be in the "code segment". There is also a "data segment"; we'll discuss these segments in a later chapter. Roughly speaking, since we are writing code, we thus do the work in the code segment. The "ORG" directive allows the programmer to place the code in different sections of program memory[7]. The assembler places the first instruction following the ORG directive at the address associated with the argument of the ORG directive.

- Lines (14-18) provide a section in which you can make your "initializations". Most embedded systems programs will have a finite amount of "stuff you need to do before your main code executes"; putting all that stuff in one section and labeling it as initialization makes for maintainable source code.

- Line (17) is the first RAT instruction in the source code. The "CLI" is a RAT instruction that instructs the hardware to ignore interrupts; interrupts is a topic we'll discuss later. The interrupt, in theory, powers up in hardware in a disabled state, but it is good programming practice to explicitly disable the interrupt anyway. Check out the RAT Assembler Manual for a full description of the CLI instruction.

- Lines (20-23) contain what embedded systems refer to as the "main code" or "task code". Note that we nicely delineate the main code with comments.

- The main portion of this program contains two lines of assembly code. Line (21) shows the first instruction, which is a bitwise logical OR instruction. The OR instruction directs the hardware to OR the hexadecimal value "0x00" with the contents of register r0 and store the result in register r0; the RTL for this operation is: r0 ← r0 OR 0x00. This instruction effectively does nothing (recall that ORing a value with '0' does not change that value) and is commonly known as a "nop" (which is short for "no operation")[8]. Check out the RAT Assembler Manual for a full description of the OR instruction.

- Line (21) also contains what we refer to as a label. Labels are important and common in assembly languages, so it is important to understand what they do. The assembler associates a number with the label; the number is the location in program memory of the instruction associated with that label. The OR instruction will reside at program memory location 0x001 (the CLI instruction resides at 0x000 based on the previous ORG directive. Labels are text that the programmer assigns. You should attempt to make labels "self-commenting"; note that the label "main" indicates that this is the main code.

- Line (22) contains a "BRN" instruction, where BRN is a mnemonic standing for "branch". This is an unconditional branch instruction which directs program flow back to the OR instruction. Thus, the BRN instruction directs the program to execute the instruction associated with the main label. Stated differently, the BRN instruction directs the program to continue at the address associated with the "OR" instruction. The execution of this program is essentially an endless loop that executes the two instructions. Nothing too exciting. Check out the RAT Assembler Manual for a full description of the BRN instruction.

[7] There are great reasons for doing this; we'll discuss those later also.
[8] The OR instruction is not actually a "nop", as a true nop does not change the state of any part of the MCU. The OR instruction can potentially change the condition flags (more on that later).

- One last thing to notice about this program is that has a high-level of organization. Even if the program did not work correctly, you could pat yourself on the back for writing such beautiful source code. This organized appearance is a result of aligning the instruction mnemonics, aligning the comments, delineating the comments, and separating major blocks of code. You should strive to do this in all of your assembly language programs.

Example 12.2: The Second RAT Program

Write a RAT program that reads in the port associated with the switches, complements the switch data, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0x02 and the LEDs are associated with port 0x0C.

Solution: Figure 12-3 shows the solution to this example. Some notes and explanation regarding some of the more interesting features of the solution follow Figure 12-3. We'll not repeat the comments we made in the previous example.

```
(00)
(01)    ;--------------------------------------------------------------------
(02)    ;- Programmer: Pat Smith
(03)    ;-
(04)    ;- Description: This program is an example RAT assembly language
(05)    ;-  program; it does very little as it is our first program.
(06)    ;--------------------------------------------------------------------
(07)
(08)    ;--------------------------------------------------------------------
(09)    ;- Assembler Directives
(10)    ;--------------------------------------------------------------------
(11)    .EQU SWITCHES = 0x02          ; input port for switches
(12)    .EQU LEDS     = 0x0C          ; output port for LEDs
(13)    ;
(14)    ;--------------------------------------------------------------------
(15)    .CSEG                         ; indicates code segment
(16)    .ORG   0x00                   ; sets the code segment counter to 0x00
(17)    ;--------------------------------------------------------------------
(18)
(19)
(20)    ;--------------------------------------------------------------------
(21)    ;- Initializations
(22)    ;--------------------------------------------------------------------
(23)             CLI                      ; prevent interrupts
(24)    ;--------------------------------------------------------------------
(25)
(26)    ;--------------------------------------------------------------------
(27)    main:       IN      r2, SWITCHES   ; put switch status into reg r2
(28)                EXOR    r2, 0xFF       ; complement switch data (trick)
(29)                OUT     r2, LEDS       ; write data to LEDs
                    BRN     main           ; main loop (do it again)
        ;--------------------------------------------------------------------
```

Figure 12-3: Solution to Example 12.2

- Lines (10-11) show a usage of the "EQU" assembler directive to provide a direct translation from the hexadecimal number to a more meaningful word. We generally consider the word to be a label, but it is a different type than the "main" label-type. This directive instructs the assembler to replace the label associated with the directive with the numerical value associated with the directive each time it occurs in the assembly source code (keeping in mind that the assembler ignores comments). The actual port addresses are a function of how the underlying hardware configuration; the assumption

here is that someone previously set up the hardware and provided the port addresses to you the programmer.

- Lines (25-28) show the main code. In words, the program repeatedly inputs the value associated with the switches, compliments that value, and then outputs that value to the LEDs. Line (25) shows the input instruction where the hardware places the value currently associated with the switches into register r2[9].

- Line (26) is a bitwise EXOR instruction that exclusive ORs the contents of register r2 with 0xFF, which effectively performs a "bit-wise"[10] compliment of the contents of r2. The EXOR instruction compliments the data in the r0 register and places that data back into the r0 register. If you were to look through the RAT instruction set, you wouldn't find an instruction that only does a compliment function. As is typical when working with assembly languages, you need to be creative with the instruction set and find a way to obtain the functionality you want with the instructions in the instruction set. The use of the EXOR instruction for a compliment is a standard assembly language programming trick, which works because EXORing a bit with a '1' effectively performs a compliment.

- Line (27) is an OUT instruction, which takes the contents of register r2 and outputs it to the port associated with the LEDs. Once again, we assume some other entity configured the hardware for you and told you which port is associated with the LEDs.

- The BRN instruction causes program control to unconditionally continue program execution at the instruction associated with the "main" label.

---

[9] We're assuming there are eight switches on the associated hardware as the RAT registers are 8-bit wide.
[10] The notion of "bit-wise" means that every bit in the register is individually EXORed with a '1'. Recall that an EXOR operation is only defined for two bits (two inputs bit with one output bit).

12.6 Chapter Summary

- There are three main types of I/O in MCUs: 1) programmed, interrupt, and DMA (direct memory access). There are two types of programmed I/O: 1) port-mapped, and, 2) memory mapped.

- There are three main parts of an assembly language program: 1) comments, 2) assembler directives, and 3) the assembly source code. Comments are messages from the programmer to other humans attempting to understand the code. Assembler directives are message from the programmer to the assembler. The assembly source code is messages from the programmer to the underlying computer hardware.

- We can divide the source code for any given assembly language program into various sections, which are standard in embedded systems programming. The two sections discussed in this chapter are, 1) the initialization code, and, 2) the main code. Every assembly language program should have these two sections clearly labeled.

- Meaningful assembly source code is neat, structured, and highly organized. It's easy to write crappy assembly language code, but a much better idea is to follow some basic formatting rules to make you source code highly readable and understandable. One the best approaches to generating good source code is to use comments to describe what you're doing and delineate different sections of the code. All languages have associated style-files that show what good assembly code looks like; be sure to access the style-file associated with any assembly code you work with.

- Assembly code is often associated with embedded systems programming. In typical embedded systems, the associated program never terminates. Likewise, the RAT instruction set has no instruction that stops execution of any running program.

- The RAT instructions that handle I/O are the IN instruction (for input) and the OUT instruction (for output). What makes computers useful is that they can communicate with the outside world; they do this via some form of I/O associated with the computer hardware in conjunction with external circuitry.

12.7 Chapter Exercises

1) List and briefly describe the three parts of an assembly language program.

2) An assembly langue program must include assembly code; briefly describe the main purpose of the other two parts of an assembly language program.

3) The three parts of an assembly language programs provide "messages" to various entities. Briefly describe those entities and the associated messages.

4) What is the first comment that every assembly language source code file should contain.

5) Briefly describe why it is important to have an area in your program for "initialization code".

6) Briefly describe why it is that embedded systems program rarely terminate unless you power-down the hardware.

7) What is the total number of input and output ports available to the RAT MCU?

8) Briefly describe whether it is possible to have both an input port and an output port have the same port_id.

9) Write a RAT program that will read in the port associated with the switches, AND the switch data with 0x34, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0xF1 and the LEDs are associated with port 0x99.

10) Write a RAT program that will read in the port associated with the switches, OR the switch data with register r30, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0x51 and the LEDs are associated with port 0x19.

11) Write a RAT program that will read in the port associated with the switches, exclusive OR the switch data, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0x34 and the LEDs are associated with port 0x39.

12) Write a RAT program that will read in the port associated with the switches, complement the switch data, OR the switch data with register r12, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0x10 and the LEDs are associated with port 0x1E.

13) Write a RAT program that will read in the port associated with the switches, exclusive NOR the switch data, and output the result to the port associated with the LEDs. Assume the switches are associated with port 0xAE and the LEDs are associated with port 0x78.

13 RAT Assembly Language Programming

13.1 Introduction

Once again, there are many approaches to teaching assembly language programming. The approach we take in this chapter is to introduce assembly language programming in the context of the RAT architecture and general assembly language programming concepts. Once you place the material in this chapter into your assembly bag of tricks, we'll be doing example problems exclusively in the next chapter. The notion here is that we'll introduce the material to you in a gentle manner in this chapter, and then show the material actual programming contexts in the next chapter. The bad news for the programmer is that there is a significant amount of material in this chapter; the good news for the programmer is that this is essentially all there is to know.

Main Chapter Topics

- **PROGRAM FLOW CONTROL:** This chapter introduces program flow control instructions including branch-type instructions and subroutines.
- **SCRATCH RAM ACCESS:** This chapter introduces instructions that directly access the scratch RAM, which includes load and store-type instructions and PUSH and POP instructions as well as LD and ST instructions.
- **INTERRUPTS:** This chapter presents a programmer's view of interrupts and provides all the information a programmer should know regarding interrupts.
- **STANDARD PROGRAMMING CONSTRUCTS:** This chapter formally introduces several the notion of three types of common constructs including iterative and if/then/else.
- **RAT MCU INSTRUCTIONS :** This chapter provides a description of all RAT MCU instructions that were not described earlier.

Why This Chapter is Important

This chapter is important because it describes some of the basic programming issues and the various RAT MCU instruction set.

13.2 Number Crunching Instructions

A majority of the RAT MCU's instruction involve the crunching of data currently store in the register file. The general model[1] for this type of instructions is that the register file sends data to the number-crunching hardware; the number-crunching hardware modifies the data, and then the RAT MCU stores the result back into the register file. The RAT MCU supports 30-types/forms of number crunching instructions; each of these instructions shares the following characteristics:

[1] This model is not 100% true for every case, so be sure to read the fine print of each instruction.

- Each instruction/form has one or two operands; one operand is always a register.
- A register is always the operand for single operand instructions. Two-operand instructions can either have two register operands or a register and an immediate value.
- We typically designate the two operands as the source operand and the destination operand. The destination operand is defined as the operand that indicates which register value can change as a result of executing a given instruction. This definition is important because it becomes confusing which operand is the source operand when there is only one operand. In cases where the instruction format only requires one operand, that operand is the destination operand according to our definition.
- When both operands are registers, the instruction can use the same register for both operands.
- The source and destination operands are usually the right operand and left operand for most RAT MCU instruction formats[2].
- Each number-crunching instruction affects the condition flag values in different and often arbitrary ways.

Table 13.1 shows the general form of the two types of number-crunching instructions. Note that all instructions include the Rd register, which is shorthand notation to mean that the instruction can designate any of the RAT MCU's 32 general-purpose registers for the operation. The column labeled "Instruction RTL" uses the word "Operation" in the notation, which is shorthand to represent one of the various operations associated with number-crunching-type instructions (for example: ADD, SUB, etc.).

Instr Type	Instruction Form	Instruction RTL
One Operand	`INSTR Rx, Ry`	Rd ← Operation(Rd, Rs)
Two Operand	`INSTR Rx`	Rd ← Operation(Rd)

Table 13.1: One & Two operand forms of number-crunching instructions.

13.2.1 The Condition Flags

The RAT MCU's number crunching instructions also contain a by-product that forms the basis of most assembly languages. The general idea is that number-crunching operations alter the value of a given register; the MCU is generally interested in various aspects regarding the changed value. The RAT MCU has two condition flags that serve to indicate a given characteristic of the altered value: the C (carry) and Z (zero) flags. Certain number-crunching operations alter these two flags in various ways, but not always in same ways; this means you have to refer to the MCU's documentation to know what each instruction does (or does not do) to the condition flags.

The two flags are actually two bits stored in the RAT MCU hardware. We refer to these two bits as the condition flags, or individually as the C flag and the Z flag. The "C" stands for "carry", so we sometime refer to this bit as the "carry flag". Likewise, the "Z" stands for "zero" so we sometime refer to this flag as the "zero flag". Number crunching instructions are the only RAT MCU instructions that have the ability to change the condition flags. The condition flags are integral to the basic operation of assembly languages. The values of the C and Z flag are set and cleared based on the results of executing some of the number crunching instructions. Some number-crunching instructions alter both the C and Z flag, some alter only the C flag or only the Z flag, and one instruction does not alter either flag.

[2] This is not true for all RAT MCU instructions; see the RAT MCU assembler manual for details. Additionally, the notion of "left" and "right" is completely arbitrary; the MCU designer chose this format for not apparent reason.

The Z flag indicates the same condition for each instruction that can alter it. The Z flag indicates that the result of the operation associated with a given instruction is either zero (all 0's), or non-zero. Specifically, then the result is zero, the Z flag is set (Z = '1'); otherwise it is cleared (Z = '0). This description fits for every instruction that tweaks the Z flag. The C flag, on the other hand, indicates different characteristics based on the given instruction. We always refer to this flag as the "carry" flag, but it only operates as a true carry bit for addition and subtraction-type instructions in the RAT MCU. We'll provide more details regarding the C and Z flag when we discuss the individual instructions that tweak them.

I personally don't memorize which instructions change the C and Z flags and how exactly those instructions change them. Although there are similarities in how the instructions tweak these flags, the differences are enough to confuse my little brain. My solution is to always have a copy of the RAT MCU assembly language manual or cheatsheet in front of me when I write assembly language code. In this way, I don't have to memorize stuff that I'll soon forget anyways. The astute programmer always refer to the instruction set manual to obtain the details of how individual instructions affect the C and Z flags before you attempt to code something important. I always allow my students to use a copy of the manual and cheatsheets when I thrash them with exams; hopefully your instructor does the same.

There are usually a few sticking points regarding the C and Z flags. The blurb below clarifies some of these points.

- There is only one C flag and one Z flag. This may seem strange in that there are 32 general-purpose registers in the RAT MCU, but this will make more sense as you start working with actually assembly language programs.
- Only certain instruction can change the C and Z flags. If you execute an instruction that does not affect the C and/or Z flag, the value of that flag does not change.
- Values of the C and Z flags don't necessarily change when the RAT MCU executes instructions that change them. The flags, however, are officially "updated" by those instructions[3].

13.2.2 Data Transfer Instruction: MOV

The move instruction (MOV) places data in to a single register (the destination register) from a single source (the source register). There are several ways to store data into registers including copying the data from the external world (IN instruction) or copying data from modules internal to the RAT MCU (LD & POP instructions). The MOV instruction is generally associated with register-to-register copying of data. The MOV instruction is the only number-crunching instruction that does not tweak either the C or Z flag. The MOV instruction has two purposes:

1) Copies data from one register (source) to a register[4] (destination) without modifying the source register.
2) Places an immediate value (source) into a register (destination).

Table 13.2 shows the two forms of the MOV instruction. The immed_val is the source for the reg-immed form. Figure 13-1 shows a well-commented code fragment demonstrates the use of both forms of the MOV instruction.

[3] The use of the word "updated" here means the "new" C and or Z value reflects the characteristics of the given operation whether the instruction altered the value of the flag or not.
[4] Generally we copy data from one register to another, but we can also copy data from one register to itself. While this sounds totally useless, it actually is one of two "pure" forms of a "nop".

Instr Type	Instruction Form	Instruction RTL	Affected Flags
reg-reg	**MOV Rd,Rs**	Rd ← Rs	none
reg-immed	**MOV Rd,immed_val**	Rd ← immed_value	

Table 13.2: The two forms of the MOV instruction.

```
(00)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)            MOV    r1,r30       ; Copies value from register r30
(02)                                ;  to register r1. The instruction
(03)                                ;  does not change the value in
(04)                                ;  register r30. Value in r1 is
(05)                                ;  overwritten by this instruction
(06)
(07)            MOV    r22,0x4A     ; Places the value0x4A into register
(08)                                ;  r22. Value in r22 is
(09)                                ;  overwritten by this instruction
(10)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-1: A usage example for both forms of the MOV instruction.

13.2.3 Logic Instructions: AND, OR, EXOR, TEST

We grouped these four instructions together because they are all logic-based instructions. As you may guess, the AND, OR and EXOR instructions are logic-based, but less obvious is the fact that the TEST instruction is also logic-based. All of these instructions perform what we refer to as *bit-wise* logic operations. This is a common notion in computerland; it simply means that given logic operator performs the logic operation on the corresponding individual bits of the two operands. In other words, the MCU operates on the MSB of the two operands, as well as every bit in-between.

Table 13.2 shows the two forms of the AND, OR, EXOR, and TEST instructions. As with the MOV instruction, the immed_val is the source for the reg-immed form. Note that the MCU automatically clears the C flag when it executes one of these instructions. The notation in Table 13.2 also indicates that the Z flag indicates if the result of the operation is zero. As you can see from comparing the RTL equations for these instructions, the hardware does not write back the result of the TEST operation to the register file (we'll cover this in more detail later). Figure 13-1 shows a well-commented code fragment that demonstrates the use of both forms of the AND, OR, and EXOR instructions.

Instr Type	Instruction Form	Instruction RTL	Affected Flags
reg-reg	**AND Rd,Rs**	Rd ← Rd · Rs	C (cleared) Z
reg-immed	**AND Rd,immed_val**	Rd ← Rd · immed_value	
reg-reg	**OR Rd,Rs**	Rd ← Rd + Rs	C (cleared) Z
reg-immed	**OR Rd,immed_val**	Rd ← Rd + immed_value	
reg-reg	**EXOR Rd,Rs**	Rd ← Rd EXOR[5] Rs	C (cleared) Z
reg-immed	**EXOR Rd,immed_val**	Rd ← Rd EXOR immed_value	
reg-reg	**TEST Rd,Rs**	Rd · Rs	C (cleared) Z
reg-immed	**TEST Rd,immed_val**	Rd · immed_value	

Table 13.3: The two forms associated with the four logic instructions.

```
(00)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)              MOV    r1,0xB8     ; Initialize registers
(02)              MOV    r2,0x0C     ;
(03)              MOV    r3,0x41     ;
(04)
(05)              AND    r1,0xF1     ; Operation: 0xB8 AND 0xF1
(06)                                 ;  Results: r1=0xB0, C=0, Z=0
(07)
(08)              AND    r1,r2       ; Operation 0xB8 AND 0x0C
(09)                                 ;  Results: r1=0x08, C=0, Z=0
(10)
(11)              OR     r2,0xB0     ; Operation: 0x0C OR 0xB0
(12)                                 ;  Results: r2=0xBC, C=0, Z=0
(13)
(14)              OR     r1,r3       ; Operation 0x08 OR 0x41
(15)                                 ;  Results: r1=0x49, C=0, Z=0
(16)
(17)              EXOR   r2,0xBC     ; Operation: 0xBC EXOR 0xBC
(18)                                 ;  Results: r2=0x00, C=0, Z=1
(19)
(20)              EXOR   r1,r3       ; Operation 0x08 EXOR 0x41
(21)                                 ;  Results: r1=0x49, C=0, Z=0
(22)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-2: A usage example for both forms of the AND, OR, and EXOR instructions.

13.2.3.1 The TEST Instruction

The TEST instruction is one of two number-crunching instructions that tweaks the condition flags based on the given operation, but does not write the result back to the destination register[6]. The TEST instruction is exactly the same as the AND operation except that the result is not written back to the register file. Note that MCU tweaks the Z flag based on the result of this operation; the MCU effectively discards the result because it does not write it back to the register file. This is a quite useful instruction in that it allows the programmer to check the values of a group of bits of any number (zero to eight bits to be exact). In other words, this instruction allows the programmer to determine if a given group of bits is all zeros or not. This functionality is quite useful for bit-masking, which we'll talk about later.

[5] Could not find proper XOR symbol… bummer!
[6] The other instruction is the CMP instruction.

The bottom line is that you don't ever really need to use the TEST instruction because you can use the AND instruction instead. However, in order to write great code, you need to be using the TEST and AND instructions in the proper manner. The rules very simple: if you're truly doing an AND operation where you need the result, use the AND instruction. If you're checking to see if some bits are set or not, use the TEST instruction. The notion here is that when you use AND instruction, someone reading your code wants to have the idea that you're doing a logical AND; that person will then look for how you use the result the AND operation. If your code does not use that result (meaning you used a AND instruction to check some bit values), that person will find you and put their shoe in your arse. This is a subtle but important difference. Assembly language programs quickly become large and hard to understand if you don't follow these rules.

```
(00)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)            MOV    r10,0xF9     ; Initialize registers
(02)            MOV    r20,0x0A     ;
(03)
(04)            TEST   r10,0x06     ; Operation: 0xF9 AND 0x06
(05)                                ;  Results: r10=0xF9, C=0, Z=1
(06)
(07)            TEST   r10,r20      ; Operation 0xF9 AND 0x0A
(08)                                ;  Results: r10=0xF9, C=0, Z=0
(09)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-3: A usage example for both forms of the TEST instruction.

13.2.4 Arithmetic Instructions: ADD, ADDC, SUB, SUBC, CMP

The arithmetic-type instructions perform the basic mathematical operations of addition and subtraction. Like many simple MCUs, the RAT MCU only has a bare minimum of arithmetic instructions, namely addition and subtraction. If you need to do more complex math such as multiplication and division, you need to use the addition and subtraction instructions to do so[7]. All is not lost, however; the RAT MCU does contain the ADDC and SUBC instructions that you can use to extend its arithmetic abilities.

Before we continue, we must mention an underlying characteristic of the RAT MCU. The RAT MCU only understands unsigned binary numbers. If you need to work with any other form of numbers, such as radix complement (2's complement), you need to work out the details in your program's code. Keep this idea in mind when you read about the RAT MCU's available arithmetic instructions. There are other processors that support arithmetic operations with other number forms, but the RAT MCU is not one of them. Bummer!

The commonality between these five instructions is that they all use the Z flag in a similar manner: the Z flag sets to indicate when the result of the given operation is zero. The ADD and SUB instructions use the C flag differently. For the ADD and ADDC, instructions, the C flag is a true "carry" flag, and indicates when the result of the addition operation exceeds an 8-bit register width. In this way, the C flag becomes the ninth bit. Stated another way, when the addition operation results in a carry, the C flag sets; otherwise it clears.

For subtraction operations, the C flag is officially functions as a "borrow" flag. This means that the C flag sets when the value that is being subtracted (subtrahend) is greater than the value that is being subtracted from (minuend). In this case, as with subtraction you perform on paper, you need to "borrow" a '1' from the next highest bit (the C flag in this case). The C flag sets to indicate this condition. This functionality is quite useful when you need to establish an equality relationship between two numbers.

The RAT MCU also includes an ADDC and SUBC instruction. These two instructions include the C flag in the given operation. Specifically, the RAT MCU adds the value of the C flag to the two operands for the ADDC instruction while the RAT MCU subtracts the value of the C flag from the minuend for the SUBC instruction. Even though these two instructions include the C flag in their operations, the RAT MCU updates

[7] The notion here is that multiplication is repeated addition and division is repeated subtraction.

the C flag to indicate characteristics about the result of the operation. These two instructions can be very useful in many cases. The most obvious case is to *easily* extend the bit-width of the addition and subtraction operations. Recall that all number-crunching operations happen on 8-bit wide registers; if you need an extended bit-range for your operations, the ADDC and SUBC support operations on larger values.

Instr Type	Instruction Form	Instruction RTL	Affected Flags
reg-reg	**ADD Rd Rs**	Rd ← Rd + Rs	C, Z
reg-immed	**ADD Rd,immed_val**	Rd ← Rd + immed_value	
reg-reg	**ADDC Rd,Rs**	Rd ← Rd + Rs	C, Z
reg-immed	**ADDC Rd,immed_val**	Rd ← Rd + immed_value + C	
reg-reg	**SUB Rd,Rs**	Rd ← Rd - Rs	C, Z
reg-immed	**SUB Rd,immed_val**	Rd ← Rd - immed_value	
reg-reg	**SUBC Rd,Rs**	Rd ← Rd – Rs – 1	C, Z
reg-immed	**SUBC Rd,immed_val**	Rd ← Rd - immed_value – C	
reg-reg	**CMP Rd,Rs**	Rd – Rs	C, Z
reg-immed	**CMP Rd,immed_val**	Rd – immed_value	

Table 13.4: The forms associated with the five arithmetic instructions.

```
(00)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)            MOV    r1,0x22     ; Initialize registers
(02)            MOV    r2,0x02     ;
(03)            MOV    r3,0x41     ;
(04)
(05)            ADD    r1,0x33     ; Operation: 0x22 + 0x33
(06)                                ;  Results: r1=0x55, C=0, Z=0
(07)
(08)            ADD    r1,r2       ; Operation 0x55 + 0xBB
(09)                                ;  Results: r1=0x00, C=1, Z=1
(10)
(11)            ADDC   r1,0x01     ; Operation: 0x00 + 0x01 + 1
(12)                                ;  Results: r2=0x02, C=0, Z=0
(13)
(14)            SUB    r1,0x01     ; Operation: 0x02 - 0x02
(15)                                ;  Results: r1=0x01, C=0, Z=0
(16)
(17)            SUB    r1,r2       ; Operation 0x01 - 0x02
(18)                                ;  Results: r1=0xFF, C=1, Z=0
(19)
(20)            SUBC   r4,0x40     ; Operation: 0x41 - 0x40 - 1
(21)                                ;  Results: r2=0x00, C=0, Z=1
(22)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-4: A usage example the arithmetic instructions ADD, ADDC, SUB, SUBC, and CMP.

13.2.4.1 The CMP Instruction

It is well known that instructions such an ADD and SUB (subtract) tweak the condition flags based on the results of these operations. In addition, these instructions and similar number crunching instructions, also necessarily write the result of the operation back to the destination register. However, as you'll see, you often

times need to check a value or result without changing the destination register. In this case, you need to use the CMP instruction.

The CMP instruction comes in two forms: there is a reg/reg form and a reg/immed form. The CMP instruction is exactly the same as an SUB instruction, but the RAT MCU does not write back the result of the subtract operation associated with the CMP instruction the destination register (register file). The RAT MCU updates the C and Z flags exactly the same as the SUB instruction, but it then discards the result of the operation. This is a very useful instruction and we typically use it in assembly language to establish a relationship between two number (<, =, or >). We'll talk more about his later in this chapter.

Once again, the bottom line is that you don't ever really need to use the CMP instruction because you can use the SUB instruction instead. However, in order to write really great code, need to be using the CMP and SUB instructions in the proper manner. The rules are very simple: if you're truly doing a subtraction operation where you need the result, use SUB. If you're comparing two values in order to establish an equality relationship, you should be using the CMP instruction. The notion here is that when you use SUB, someone reading your code wants to have the idea that you're doing subtraction; that person will then be on the lookout for how you use the result. If your code does not use that result (meaning you used a SUB instruction for a comparison operation), that person will hire a maniacal clown to attack you during the night when you're sleeping[8]. This is a subtle but important difference. Assembly language programs quickly become large and hard to understand if you don't follow these types of guidelines[9].

```
(00)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)            MOV    r1,0x44     ; Initialize registers
(02)            MOV    r2,0x44     ;
(03)
(04)            CMP    r1,0x33     ; Operation: 0x44 - 0x33
(05)                               ;  Results: r1=0x44, C=0, Z=0
(06)
(07)            CMP    r1,r2       ; Operation 0x44 - 0x44
(08)                               ;  Results: r1=0x44, C=0, Z=1
(09)
(10)            CMP    r1,0x55     ; Operation 0x44 - 0x55
(11)                               ;  Results: r1=0x44, C=1, Z=0
(12)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-5: A usage example the CMP instruction.

13.2.5 Shift & Rotate Instructions: LSL, LSR, ROL, ROR, ASR

The shift & rotate instruction perform simple shift operations on the bits within registers. While we can consider these shift-type operations as a form of mathematical operations, we're opting to group them separately from the arithmetic instructions[10]. Each of these instructions is reg-type instructions and thus only has one operand, which we consider the destination operand. Thus, the source and destination operands for the shift & rotate instructions are the same.

The shift & rotate instructions use the Z flag to indicate whether the result of the operation is zero or not. The C flag is slightly more complicated. Where the C flag goes and what it becomes in these instructions is completely arbitrary; that is to say, the computer design could do anything with it, but they for some reason decided (either due to excessive cannabis intake, or lack thereof) to assign them the way they did for the RAT MCU. It's too klunky to list how these instruction handle the C flag here; be sure to check out the notational descriptions in Table 13.5 and Table 13.6.

[8] The clowns will attack you in the daytime if you're an engineering major, because they know you don't have time to sleep.
[9] There are many guidelines such as these associated assembly language programming. We'll list them all as the come up in the various programming examples in this text.
[10] Recall that a single bit-level shift-left and shift-right operations are clever ways to perform a multiply by 2 or divide by 2, respectively.

Instr Type	Instruction Form	Instruction RTL	Affected Flags
reg	**LSL Rd**	Rd ← Rd(6:0) & C	C←MSB, Z
reg	**LSR Rd**	Rd ← C & Rd(7:1)	C←LSB, Z
reg	**ROL Rd**	Rd ← Rd(6:0) & Rd(7)	C←MSB, Z
reg	**ROR Rd**	Rd ← Rd(0) & Rd(7:1)	C←LSB, Z
reg	**ASR Rd**	Rd ← Rd(7) & Rd(7) & Rd(6:1)	C←LSB, Z

Table 13.5: The forms associated with the five shift & rotate instructions.

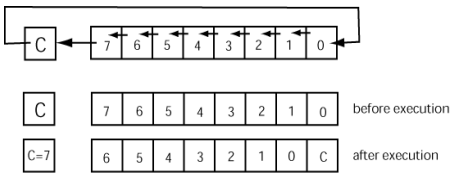
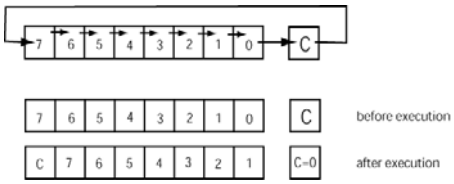
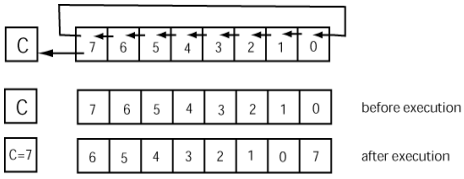
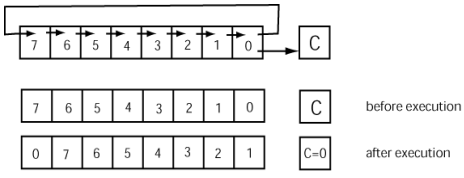
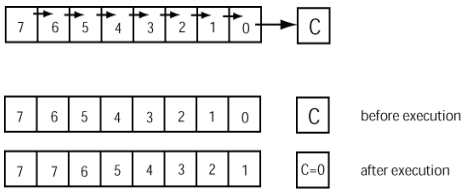
Instr Type	Visual Description	Verbal Description
LSL		• All bits of register are shifted to the left • The previous C flag becomes the new LSB • The previous MSB becomes the new C flag
LSR		• All bits of register are shifted to the right • The previous C flag becomes the new MSB • The previous LSB becomes the new C flag
ROL		• All bits of register are shifted to the left • The previous MSB becomes the new LSB • The previous MSB becomes the new C flag
ROR		• All bits of register are shifted to the right • The previous LSB becomes the new MSB • The previous LSB becomes the new C flag
ASR		• All bits of register are shifted to the right • The previous LSB becomes the new C flag • The previous MSB does not change

Table 13.6: Visual and written descriptions of the five shift & rotate instructions.

Figure 13-6 shows a code fragment that exercises each of the shift & rotate instructions. Figure 13-6 implicitly shows one of the important issues that you need to consider when dealing with the LSR & LSL instructions. Because the C flag value shifts into the resulting register value, you must know what the value of the C flag is before you issue a LSL or LSR instruction. The code in Figure 13-6 set the C flag using a subtraction operation, but this is easier to do by using a SEC instruction (we'll describe the SEC instruction in a later section of this chapter.

```
(00)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)  MOV    r0,0x11     ; Initialize registers
(02)           MOV    r1,0x23     ; Initialize registers
(03)           MOV    r2,0x03     ;
(04)           MOV    r3,0x82     ;
(05)           MOV    r4,0x8C     ;
(06)           MOV    r5,0x86     ;
(07)           SUB    r0,r1       ; set carry flag
(08)
(09)           LSL    r1          ; Operation: r1=0x23 (C=1)
(10)                               ;  Results:  r1=0x91, C=1, Z=0
(11)
(12)           LSR    r3          ; Operation: r3=0x82 (C=1)
(13)                               ;  Results:  r3=0x05, C=1, Z=0
(14)
(15)           ROL    r2          ; Operation: r2=0x03
(16)                               ;  Results:  r2=0x06, C=0, Z=0
(17)
(18)           ROR    r4          ; Operation: r4=0x02
(19)                               ;  Results:  r4=0x19, C=1, Z=0
(20)
(21)           ASR    r5          ; Operation: r5=0x86
(22)                               ;  Results:  r5=0xC3, C=0, Z=0
(23)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-6: A usage example the five shift & rotate instructions.

13.3 Program Flow Control

Our original definition of a computer was a "piece of hardware that sequentially executes a stored program". The key word here is sequentially. Computer programs typically execute programs in sequence: one instruction after another. Recall that program memory stores the instructions, which means that sequential program execution essentially means that computers execute the instruction at one memory address, and then the computer executes the instruction at the next contiguous memory address, etc. As you will start to see, programs do not always simply execute instructions sequentially; if they did so, the program will quickly run out of instructions to execute.

The notion of "program flow control" deals with instructions that have the ability to cause the computer to execute instructions in some order other than strictly sequentially. In other words, some instruction instruct the CPU "jump" somewhere in program memory other than to the instruction following the current instruction. We consider sequential instruction execution as "normal operation", while everything else falls into the category of non-normal operation, or more aptly put, program flow control. This section covers the three areas of instructions and concepts associated with program flow control: 1) branch instructions, 2) subroutines, and 3) interrupts.

13.3.2 Branch Instructions

Branch instructions *can*[11] cause the MCU to execute an instruction that is not the next instruction in program memory. There are two types of branch instructions: unconditional branches and conditional branches. Both types of branch instructions potentially alter the sequence in which the MCU executes instructions from program memory. The difference between these two types of instructions is that unconditional branches

[11] We use the word "can" here because some types of branch instructions don't always branch.

always change the program execution sequence while conditional branches may or may not change the instruction execution sequence depending on certain conditions in the MCU. The main thing to keep in mind is that branch instructions have the ability to transfer program control from one instruction to another instruction that is not necessarily the next instruction in the sequence.

13.3.2.1 The Unconditional Branch Instruction: BRN

As the name implies, when the MCU executes an unconditional branch instruction, the MCU always takes the branch. In other words, program control always transfers to another instruction in program memory that is not the instruction following the instruction just executed[12]. The BRN instruction implements an unconditional branch for the RAT MCU. Figure 13-7 shows an example of an unconditional BRN instruction. Here are some rather important things to notice about program in Figure 13-7.

- The program does not do anything useful; it only serves as an example of a BRN instruction. The reality is that IN, EXOR, and OUT instructions are placeholders for something important that would be going on in a real program. Not surprisingly, this example is similar to one of our first example programs in a previous chapter.

- At a high level, the BRN instruction of Figure 13-7 simply transfers program control to the instruction that follows (on the same line) the label "main". At a lower level, here is what really goes on. The assembler replaces the label "main" (the single operand in the BRN instruction) with the effective address in program memory of the instruction associated with the main label. In this context, the effective address refers to the location in program memory of the instruction that is associated (on the same line as) or follows (the next valid instruction after the label in cases where the label and instruction do not appear on the same line[13]). The assembler generates this effective address on the "first pass" of the assembler, specifically when the assembler first encounters the label. The assembler the substitutes the effective address for each occurrence of the word "main" when it appears as an operand of an instruction. When an instruction with "main" as an operand executes (such as with the BRN instruction and others we'll talk about soon), the assembler places the effective address associated with the "main" label into the immed-field of the branch instruction. When the MCU executes an unconditional branch instruction, the underlying hardware ensures that the MCU will next execute the instruction in program memory at the address associated with the label.

- It is always straightforward to calculate the effective addresses associated with program labels. For the program in Figure 13-7, the CSEG was "ORGed" to address 0x00 of the code segment on line (07) of the program. The first instruction in the program on line (13) therefore resides at address 0x00. Because the main label is on the line associated with the second instruction in the program, the assembler assigns the main label to a value of 0x01, as that is the location in program memory of the second instruction in the program.

- As will all RAT assembly language programs, the program in Figure 13-7 never stops executing due to the unconditional branch instruction directing program flow back to some useful place in the program.

[12] You can branch to the instruction following the branch instruction, but that generally means you'r not understanding the point of the branch instruction.
[13] Labels do not need to have instructions on the same line; labels can appear on a line with nothing else on them.

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-----------------------------------------------------------------------
(03)  .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)  .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)  ;
(06)  .CSEG
(07)  .ORG  0x00        ; memory location of instruction data
(08)  ;-----------------------------------------------------------------------
(09)  ;
(10)  ;-----------------------------------------------------------------------
(11)  ;- Initializations
(12)  ;-----------------------------------------------------------------------
(13)  init:    CLI                         ; prevent interrupts
(14)  ;-----------------------------------------------------------------------
(15)                                       ;
(16)  main:    IN       r0,DATA_IN_PORT   ; grab data, place in r0
(17)           EXOR     r0,0xFF           ; do something worthwhile
(18)           OUT      r0,DATA_OUT_PORT  ; output some data
(19)           BRN      main              ; go back to main loop
(20)  ;-----------------------------------------------------------------------
```

Figure 13-7: An unconditional branch instruction (BRN).

13.3.2.2 Conditional Branch Instructions: BREQ, BRNE, BRCC, BRCS

While the unconditional branch instructions don't provide many options in terms of program flow control, the conditional branch instructions do. Unconditional branch instructions utilize two persistent (meaning the values are stored somewhere in the underlying RAT hardware) bits. We refer to these two bits as the condition flags, or individually as the C flag and the Z flag. The "C" stands for "carry", so we sometime refer to this bit as the "carry flag". Likewise, the "Z" stands for "zero" so we sometime refer to this flag as the "zero flag". Only instructions that involve the ALU can alter the value of the C and Z flags, which means that conditional branch instructions typically follow ALU-type instructions.

The values of the C and Z flag are set and cleared based on the results of executing instructions that involve number-crunching. Some number-crunching-based instructions alter both the C and Z flag, some ALU-based instructions alter only the C flag or only the Z flag, and one does not alter either flag. For example, when the RAT MCU executes an ADD instruction, which surprisingly is an addition operation, the Z flag is set (Z='1') when the result of the operation is 0x00; the Z flag is cleared (Z='0') otherwise. Likewise, if executing the add instruction generates a carry from the 8^{th} bit of the operation, the C flag is set (C='1'); otherwise the C flag is cleared (C='0') is there is no carry.

This basic C and Z flag operation is similar for all ALU-based instructions. I truly don't memorize which instructions set or clear the condition flags, because when I write RAT assembly language programs, I always have the RAT Assembler manual in front of me. In this way, I don't have to memorize stuff that I'll soon forget anyways. The astute programmer always refer to the instruction set manual to obtain the details of how individual instructions affect the C and Z flags before you attempt to code something important.

The RAT MCU has four types of conditional branch instructions; Table 13.7 lists these types along with some important details. Once again, check the RAT assembler manual for full details. As you can see from examining Table 13.7, conditional branches based on the carry flags have more memorable mnemonics with "BRCS" for "branch carry set" and "BRCC" for "branch carry clear". The case for the Z flag is a bit more cryptic due to the nature of how the RAT MCU checks for zero. Most MCUs check for zero by subtracting the source operand from the destination operand; if the result is zero, then the two values must have been equal. In the case where they are equal, the Z flag will be set; all this results in the mnemonic of "BREQ". If the Z flag clears after the operation, the result of the subtraction was non-zero; hence, the mnemonic "BRNE" for "branch not equal".

Instruction	Comment
BREQ label	branch when zero flag set (Z=1)
BRNE label	branch when zero flag cleared (Z=0)
BRCS label	branch when carry flag set (C=1)
BRCC label	branch when carry flag not set (C=0)

Table 13.7: Overview of RAT conditional branch-type instructions.

Figure 13-8 shows a program similar to the program in Figure 13-7, except that it uses a both a conditional as well as an unconditional branch. The following are the main items of interest regarding Figure 13-8.

- Line (18) contains the conditional branch instruction BREQ. When the program executes this instruction, one of two things will happen. If the current value of the Z flag is zero (Z='0'), program control transfers to the instruction associated with the "no_out" label; otherwise, program control transfers to the next instruction in the program at line (20). And for the record, the address value associated with the "no_out" label is 0x05.

- The ADD instruction has the effect of adding 0x00 (an immediate source operand) to register r0 (the destination operand) and storing the result in r0. This instruction seems to do nothing, as adding 0x00 does not change the value of r0. In truth, this instruction actually does something significantly important. As with most instructions that utilize the ALU, the condition flags (the C and Z flags) are set based on the result of the operation requested by the instruction. In this case, since the instruction added 0x00 to a register, the C and Z flags are set based on the value in r0. Specifically, if r0 contains 0x00, the Z flag will be set after the ADD operation; otherwise it will be cleared. Ironically, the MCU will always clear the C flag in this case as adding a 0x00 to a register never generates a carry.

- If you had to describe this program to one of your non-technical friends, you would say something like this: "This program constantly monitors the input port; when the input data from that given port is 0x00, the program writes the data to the output port". Almost too much excitement for one program.

```
(00)    ;-----------------------------------------------------------------------
(01)    ;- Assembler Directives
(02)    ;-----------------------------------------------------------------------
(03)    .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)    .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)    ;
(06)    .CSEG
(07)    .ORG  0x00        ; when in program memory to place program
(08)    ;-----------------------------------------------------------------------
(09)    ;
(10)    ;-----------------------------------------------------------------------
(11)    ;- Initializations
(12)    ;-----------------------------------------------------------------------
(13)    init:      CLI                        ; prevent interrupts
(14)    ;-----------------------------------------------------------------------
(15)                                          ;
(16)    main:      IN       r0,DATA_IN_PORT   ; grab data, place in r0
(17)               ADD      r0,0x00           ; tweak Z flags; clear the C flag
(18)               BREQ     no_out            ; branch if r0=0; skip OUT instr
(19)                                          ;
(20)               OUT      r0,DATA_OUT_PORT  ; output some data
(21)    no_out:    BRN      main              ; go back to main loop
(22)    ;-----------------------------------------------------------------------
```

Figure 13-8: A conditional branch instruction (BREQ).

Figure 13-9 rewrites the example from Figure 13-9 using a CMP instruction rather than an ADD instruction. In particular, the program in Figure 13-9 uses the reg-immed form of the CMP instruction. The important thing to notice about the program in Figure 13-9 is that it is equivalent to the program in Figure 13-9. However, the program in Figure 13-9 is a significantly better approach. Why is it a better approach? Because when you use the CMP instruction, it provides a message to any human reading the program that you're comparing two values, most likely in hopes of establishing a relationship between those two number (<, =, or >). This a much clearer approach than using the ADD instruction (or a SUB instruction for that matter) as the use of an ADD instruction should be a message to the human reader of your program that you're intending on adding two numbers. This is a subtle but important difference. Assembly language programs quickly become large and hard to understand if you don't follow these types of guidelines[14].

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)  .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)  ;
(06)  .CSEG
(07)  .ORG  0x00        ; memory location of instruction data
(08)  ;-------------------------------------------------------------------------
(09)  ;
(10)  ;-------------------------------------------------------------------------
(11)  ;- Initializations
(12)  ;-------------------------------------------------------------------------
(13)  init:      CLI                         ; prevent interrupts
(14)  ;-------------------------------------------------------------------------
(15)                                         ;
(16)  main:      IN       r0,DATA_IN_PORT    ; grab data, place in r0
(17)             CMP      r0,0x00            ; tweak the C and Z flags
(18)             BREQ     no_out             ; branch if r0=0; skip OUT instr
(19)                                         ;
(20)             OUT      r0,DATA_OUT_PORT   ; output some data
(21)  no_out:    BRN      main               ; go back to main loop
(22)  ;-------------------------------------------------------------------------
```

Figure 13-9: A conditional branch instruction using a CMP instruction to set the Z flag.

13.3.3 Conditional Branch Constructs

Several types of common programming constructs use conditional branch instructions. These main types of constructs include 1) *if/else* constructs, 2*) iterative loop* constructs, and 3) *conditional loop* constructs. These three constructs represent assembly language versions of constructs that form structured programming[15]. We use these three constructs extensively in assembly language programming. In fact, we use these constructs so often that we consider good assembly language programs as being built around these basic constructs. You'll see these constructs in all of the examples that follow. The examples that follow present each of these constructs in assembly language program setting. Though these examples that follow lack true substance, they do show how the RAT MCU assembly language implements these basic constructs.

13.3.3.1 If/Else Constructs

Figure 13-10 shows an example of an *if/else* construct. As the names implies, the code will either do one thing (if some condition is met), or *else* it will do some other thing (if the original condition is not met). In the case of the code fragment in Figure 13-10, bases the condition that may or may not be met on the state of the Z flag

[14] There are many guidelines such as these associated assembly language programming. We'll list them all as the come up in the various programming examples in this text.

[15] The notion of structured programming is that we can decompose all "good" programs, including assembly language programs, into just a handful of basic programming constructs. Three of those constructs are if/else, iterative loops, and conditional loops. The notion here is that if you can't decompose your programs into basic constructs such as these, then your programs are poorly structured and will eventually fail. Some people refer to programs such as these as spaghetti code.

when the MCU executes the branch instruction. The program takes one code path if the data meets the condition and another path if the data does not meet condition. Here is a full description of the program.

- Here is a high-level description of this program: "If the value read from the input port is non-zero, then output 0xFF to the output port; otherwise, output 0x00 from the output port".

- The BRNE instruction (line 17) forms the basis of the if/else construct. The "if" part of the BRNE instruction involves "if the value read from the input port is non-zero". Line (21) contains the entirety of the "if" part of the construct. The "else" part of the BRNE instruction involves "if the value read from the input port is zero". Lines (18-19) contain the entirety of the "else" part of the construct.

- Because this is an if/else construct, program execution either follows the "if" path or the "else" path. There are no other possibilities here.

- There are five labels in this program: init, main, zero, not_zero, and out_val. Note that the program itself never references the init and zero labels. Including these labels in the code is thus a form of "self-commenting" and makes the code more readable for humans who may have to read through and/or understand this code. It is always good programming practice to include as many labels as you need to make your code as readable as possible. For the record, the address values associated with the init, main, zero, not_zero, and out_val labels are 0x20, 0x21, 0x24, 0x26, and 0x27 respectively.

- We list all labels using lower case. This is once again good programming practice as you'll see later that we save upper-case labels to differentiate subroutines (a later topic in this chapter).

```
(00)   ;--------------------------------------------------------------------
(01)   ;- Assembler Directives
(02)   ;--------------------------------------------------------------------
(03)   .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)   .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)   ;
(06)   .CSEG
(07)   .ORG  0x20        ; memory location of instruction data
(08)   ;
(09)   ;--------------------------------------------------------------------
(10)   ;- Initializations
(11)   ;--------------------------------------------------------------------
(12)   init:       CLI                        ; prevent interrupts
(13)   ;--------------------------------------------------------------------
(14)
(15)   main:       IN     r10,DATA_IN_PORT    ; input some data
(16)               CMP    r10,0x00            ; set condition flags
(17)               BRNE   not_zero            ; jump if r10 is non-zero
(18)   zero:       MOV    r11,0x00            ; clear bits in r11
(19)               BRN    out_val             ; jump to output instruction
(20)                                          ;
(21)   not_zero:   MOV    r11,0xFF            ; set bits in r11
(22)   out_val:    OUT    r11,DATA_OUT_PORT   ; output some data
(23)               BRN    main                ; do it again
(24)   ;--------------------------------------------------------------------
```

Figure 13-10: Example of an *if/else* construct in RAT assembly language.

As you may be wondering, there are two different approaches to applying the same if/else construct. Consider the two fragments of code in Figure 13-11, which shows two functionally equivalent approaches to an if/else statement. The functionality the code implements is:

```
input byte;
    if byte is 0x00, then r1=0x0A;
    else                  r1=0x0B
output r1
```

Both code fragments in Figure 13-11 take the approach of testing whether the input value is zero or not. The difference in these two fragments of code is that the code in Figure 13-11(a) branches when the input value is not equal to zero while the code in Figure 13-11(b) branches when the value of the input value is zero. These two fragments of code are not exactly equivalent, but we consider them functionally equivalent because they perform the exact same function.

Regarding the code in Figure 13-11, the question arises: "Which is the better or preferred approach". In this case, as in most if/else clauses, there is arguably not a better approach. There are some cases when it is "more clear" to the human reader to code it one way or the other, but this example is too simple to be one of those cases.

(a)

```
(00)            IN     r0,IN_PORT    ; grab data
(01)            CMP    r0,0x00       ; is byte is 0x00?
(02)            BRNE   not_zero      ; jump if r0 is != 0x00
(03)            MOV    r1,0x0A       ; place 0x0A in r1
(04)            BRN    out_val       ; jump to out instr
(05)                                 ;
(06) not_zero:  MOV    r1,0x0B       ; place 0x0B in r1
(07) out_val:   OUT    r1,OUT_PORT   ; output some data
```

(b)

```
(00)            IN     r0,IN_PORT    ; grab data
(01)            CMP    r0,0x00       ; is byte is 0x00?
(02)            BREQ   zero          ; jump if r0 = 0x00
(03)            MOV    r1,0x0B       ; place 0x0B in r1
(04)            BRN    out_val       ; jump to out instr
(05)                                 ;
(06) zero:      MOV    r1,0x0A       ; place 0x0A in r1
(07) out_val:   OUT    r1,OUT_PORT   ; output some data
```

Figure 13-11: Code fragments showing two equivvalent approaches to if/else constructs.

13.3.3.2 Iterative Loop Constructs

Figure 13-12 an example of a RAT assembly language implementation of an iterative construct. Iterative loops are probably the most common constructs seen in assembly language programming, followed by the if/else construct. The main idea of an iterative construct is to put repetitive tasks into loops in an effort to save code space. Nothing prevents you from not using loops in your assembly language programs except the size of the program memory. Loops present only a small amount of processing overhead and you should use them whenever possible.

The code in Figure 13-12 shows a fragment (not a complete program) that highlights the important features of an iterative loop. The body of the loop in this example contains some pointless code that we use as a placeholder for something more meaningful. This code consists of some inputting, processing it, and then outputting a value. The idea here is that we want to do this for a set number of times and *we know* in advance how many times we want to do it (or worded differently, we know how many times we want to iterate the loop). Here are some more interesting things regarding the code of Figure 13-12.

- Line (14) shows the first line of interest in this code, which is to set the loop variable to some value. In this case, we arbitrarily use register r3 as "loop variable"; we could have chosen any unused register. We set the loop variable to a value that represents the number of times we want the loop to execute. The loop variable also goes by names such as the *iterative variable*, or the *loop counter*. Before you start any iterative loop such as this, you always must make sure to that the loop variable reflects the number of time you want to execute the body of the loop.

- Once the body of the loop executes, the code decrements the loop counter variable. We use a SUB (subtract) instruction to decrement the loop counter; using the SUB instruction has the effect of

setting the Z flag based on the result of the subtraction operation. When the loop count is non-zero, program control branches to execute the body of the loop another time (in the case of this program, program control returns to the instruction associated with the "loop" label. When the loop count is zero, the Z flag is set (as a result from executing the SUB instruction) and program control drops down to the next instruction following the BRNE instruction (the conditional jump instruction).

- We know that all loops have "loop overhead". We consider the loop overhead to be the instructions involved in making the loop "work", but at the same time, instructions that have nothing to do with what goes on inside of the loop. In this case, the loop overhead is in line (14) where we set the iterative count, and lines (19-20) where we check to see if the loop has completed its desired amount of iterations. It is because of this loop overhead that you don't want loops that iterate only once or twice; in these cases, it is better to use a set of instructions in a non-iterative form (no loop construct). We'll discuss the concept of loop overhead in a later chapter.

- There are many ways to code an iterative loop; this example shows one way. I believe this is the easiest way, which is why I use this example. The other main approach is to start the loop at 0x00 or 0x01 and increment a counter; part of the loop overhead in this case would be to check to see if the loop counter has reached its terminal count (and trying to avoid the nasty one-off error). This approach works for me and it always works, so I never attempt to do it any other way in an effort to avoid making stupid mistakes. For that matter, if you code does not work for some reason, one of the first places you should look is at the iterative constructions.

```
(00)   ;-----------------------------------------------------------------
(01)   ;- Assembler Directives
(02)   ;-----------------------------------------------------------------
(03)   .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)   .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)   ;
(06)   .CSEG
(07)   .ORG  0x40                       ; memory location of instructions
(08)   ;-----------------------------------------------------------------
(09)   ; iterative loop construct
(10)   ;-----------------------------------------------------------------
(11)
(12)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(13)   ;
(14)               MOV     r3,0x08             ; load iterative count value
(15)                                           ;
(16)   loop:       IN      r1,DATA_IN_PORT     ; grab data
(17)               EXOR    r1,0xFF             ; toggle input data
(18)               OUT     r1,DATA_OUT_PORT    ; output some data
(19)               SUB     r3,0x01             ; decrement iteration variable
(20)               BRNE    loop                ; do it again if count non-zero
(21)                                           ;
(22)   loop_done:                              ; do something else…
(23)
(24)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-12: Example of an iterative loop construct with a RAT assembly code fragment.

13.3.3.3 Conditional Loop Constructs

The final construct based on conditional branch instructions is the conditional loop. There is one simple difference between iterative and conditional loops. In iterative loops, we know the number of times that the body of the loop needs to execute (the iteration count, or loop count) prior to starting the loop. Thus, when you executed the loop the given number of times, the program no longer executes the loop. With conditional loops, we do not know the number of times the loop needs to execute prior to entering into the loop. Conditional loops terminate when the MCU hardware meets a given condition.

It should be no surprise that the MCU's condition flags are the only conditions that you could use to control the conditional loop constructs. In this way, there are only four possible scenarios for breaking out of a

conditional loop construct; the loop ends when either the C flag becomes a '1' or '0', or the of Z flag becomes a '1' or '0'.

The fragment in Figure 13-13 shows an example of a conditional loop. Here are some of the more interesting things to note regarding this code fragment

- The code in general is inputting a value, doing something to that value (the EXOR), then adding the result to a register (r1). The notion here is that we're inputting some value from the outside world and adding the value repeatedly to a register. This being the case, you would imagine that after enough non-zero adds to the register, the register would produce a carry. When the program detects a carry, program control exits the loop. You won't know how many times the code stays in the loop because you generally do not know what values will be added to the register, as they are a function of the input value.

- Many time in assembly language, you'll need to "keep track" of a many similar operations. Figure 13-13 does something similar in that is continually adds values to register r0. We typically refer to this action as "accumulation", and we speak of register r0 as being an "accumulator". In this example, you can probably see that the value in the accumulator will eventually exceed 8-bits and hence generate a carry out. You can also "accumulate" in the other direction by starting with a large number and continually subtracting. This example is arbitrary and doesn't really do anything except show the important parts of a conditional construct. Maybe most importantly here, if you're planning to use an accumulator, make sure you always first initialize the register you plan to use as the accumulator.

- In the code of Figure 13-13, the code clears the r0 register prior to entering the loop; the code uses this register as an accumulator of sorts in that it essentially accumulates results from the body of the loop. For the case of the code shown in Figure 13-13, the code continually adds values to the s0 register. The instructions continually add the values and then examines the carry flag after every addition. If the carry flag is not set, the ADD operation did not generate a carry and the loop body executes another time (another iteration). If the carry flag is set, the ADD operation caused an overflow and program control will drop through (it will not take the jump, thus exiting the iterative construct).

- In more technical terms, the code in Figure 13-13 works as follows. Line (19) sends the program control back to the instruction associated with the "loop" label as long as the BRCC instruction sees that the C flag is cleared after the add operation. Not surprisingly, the BRCC mnemonic stands for "branch when carry cleared". When the result of the ADD instruction on line (18) results in a carry out of the 8-bit r0 register, the MCU sets the C flag and program control drops through the BRCC instruction (the code does not take the branch).

- The conditional loop construct also has loop overhead. Initializing the accumulator (line 13) and the BRCC instruction (line 19) form the loop overhead. We refer to this as overhead because the instructions require time to execute, but don't do anything use other than the general accounting tasks associated with the loop. Every other instruction in this example is arbitrary; the loop construct is what interests us.

- The program fragment does not use the "loop_done" label. We include it in this example as a form of self-commenting, keeping in mind that labels are not instructions and thus do not increase the size of the program.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU  DATA_IN_PORT       = 0x56   ; example input port data
(04)  .EQU  DATA_OUT_PORT      = 0x57   ; example output port data
(05)  ;
(06)  .CSEG
(07)  .ORG  0x00                        ; memory location of instructions
(08)  ;-------------------------------------------------------------------------
(09)  ; conditional loop construct
(10)  ;-------------------------------------------------------------------------
(11)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(12)  ;
(13)              MOV      r0,0x00             ; clear reg; use as accumulator
(14)                                           ;
(15)  loop:       IN       r1,DATA_IN_PORT     ; grab some data
(16)              EXOR     r1,0xFF             ; compliment bits
(17)              OUT      r1,DATA_OUT_PORT    ; output some data
(18)              ADD      r0,r1               ; add value to accumulation r0
(19)              BRCC     loop                ; repeat is no carry generated
(20)              ;
(21)  loop_done:  ; do something else…
(22)  ;
(23)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-13: Example of a conditional loop construct in a RAT assembly code fragment.

13.3.4 Subroutines: CALL & RET

It is frequently necessary for a program to execute the same set of instructions at several different points in a program. If the set of instructions is relatively short, you could simply place the code in the program wherever your program required it. However, if this section of code is relatively long, a more effective use of program code space to have only one piece of the code that needs repeating. When that section of code needs to execute, the program control transfers to the section of code that requires execution, the program executes that code, and program control transfer back to the code that it originally transferred from. Thus, when the special section of code completes execution, program control returns to where it was before the program executed that special piece of code. The thing we are describing here is what we know in assembly language terms as a *subroutine*. We refer to this same mechanism in a higher-level language a *function* or a *method*, depending on the higher-level language you're working with.

In actuality, subroutines have another major purpose associated with them that is extremely important to assembly language programs: they give the programmer the ability to modularize and thus organize their programs. In essence, there are situations where you *must* use subroutines as they save you program code space, but there are also situations where you *should* use subroutines to keep your programs organized. We describe these ideas in more detail in a later chapter.

In the end, the notion of using subroutines in your programs can't be overemphasized. Good programmers use appropriately named and structured subroutines in order to control the complexity of their code. Conversely, you can always detect code from beginning assembly language programmers, as they tend to avoid using subroutines. The concept of subroutines from a programmer's level is straightforward; please don't fear the subroutine. The use of subroutines is important for many reasons, the main ones we list below.

- Efficient Coding: the section of code that needs executing multiple times appears only one time in the source listing only once. This represents a significant savings in program code space because despite the code appearing only once, the programmer can easily execute it multiple times. The efficiency we refer to here is program memory space efficiency, which is not always the same as run-time efficiency.

- Program Readability and Understandability: Placing large amounts of code in subroutines and giving it an appropriate name makes the program more readable. This allows the programmer to abstract the code to higher levels, which means you don't necessarily need to understand the workings of code at low level (the code in the subroutine) in order to use the code. Additionally, providing the code with a

self-commenting label (or subroutine name) allows another human to quickly understand the purpose of the code on a higher level.

- Maintainability: Compartmentalizing the code allows you to quickly locate and easily change only the code you need, particularly if it later turns out that there is a problem with that code.

- Reusability: Placing sections of code in meaningful blocks, namely subroutines, increases the chances that you or someone else can reuse the code later. This saves time by preventing multiple programmers from "re-inventing the wheel". This is also why you should always provide adequate commenting on your subroutines.

Figure 13-14 shows an example of software flow diagrams that justifies the use of subroutines. In Figure 13-14(a), program flow continues in a linear manner and executes the sections of code represented by A, B, C and X. The code represented by X appears two times in the section of code. In Figure 13-14(b), program flow jumps from A to X and then from X to B. Likewise, it jumps from B to X and then from X to C. In this way, the code that represents X needs to appear in the code only one time. This represents a direct saving in code space in program memory, though there are other negative ramifications in terms of program execution efficiencies (this is the overhead issue; we'll talk about this later). The notion Figure 13-14 is attempting to convey is that using subroutines will save program code space.

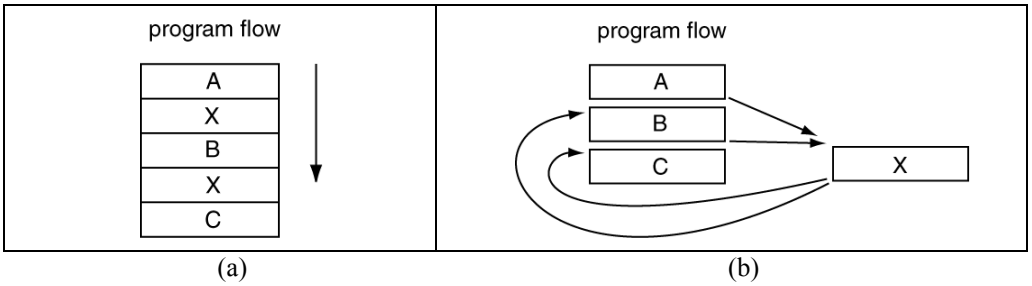


Figure 13-14: A diagram showing program flow both with and without subroutines.

Subroutines are a form of program control not unlike the unconditional branch instruction. When the program executes subroutines, the program temporarily transfers program control to some other place in the program (or more precisely, some other address in program memory). When the subroutine completes it execution, the program transfers program control back to the instruction that follows the instruction that invoked the subroutine. The instruction that invokes a subroutine is the CALL instruction. The instruction that returns control back to the instruction following the CALL instruction is a RET instruction.

A typical subroutine call in RAT assembly language looks something like this: "`CALL  my_subroutine`". Note that the CALL instruction is an immed-type instruction where the word "my_subroutine" is associated with an address value. The word "my_subroutine" is nothing more than a label that is no different from the other RAT labels we've previously discussed. Once again, when the assembler assembles the assembly code, the assembler associates an address with the "my_subroutine" label. This address is the location in instruction memory of the code that serves as the first instruction in the "my_subroutine" subroutine.

At the end of the subroutine code, there is usually some type of *return* instruction. The official instruction for the RAT assembly code is simply "RET", which is a none-type instruction since it has no operands. This instruction indicates that the program completed execution of the code associated with the subroutine and program control should now return to the instruction immediately following the after the original CALL instruction. This represents a high-level description of subroutines; a lower-level description if the CALL/RET mechanism requires interaction with the stack and we'll cover those details in a later.

Figure 13-15 shows the general form of the subroutine construct. In the program fragment of Figure 13-15, the program encounters the CALL instruction after it completes execution of the AND instruction on line (02). The CALL instruction then transfers program control to the instruction associated with the "my_sub" label, which is on line (12). The program then executes the instructions associated with and following the "my_sub" label through the RET instruction. When the program encounters the RET instruction, program control returns to the instruction following the original CALL instruction, which in this case is the OR instruction on line (03). This seems somewhat magic, but the underlying hardware handles all the details.

Note that there is an important relationship between CALL and RET instructions: For every CALL instruction there must be an accompanying RET instruction. If you violate this universal constant, your programs will not execute properly. The specific reason your programs won't operate properly is that you have violated the "correctness" or "sanctity" of the CALL/RET pairs. Once again, we'll speak more about this in a later chapter.

Figure 13-15 contain some other key details that are well worth mentioning. Here they are.

- The subroutine in Figure 13-15 has some very nice properties. First, the code nicely delineates the subroutine from other parts of the program with various clever forms of commenting. The most useful form of commenting is the fact that the subroutine has a nice banner that provides some basic information about the subroutine. This information includes the subroutine name, a description of the subroutine, and a list of registers that the subroutine modifies. This should be your standard for all subroutines

- Related to the previous bullet, you must realize that subroutines can change both register and flag values. This is true because there is only one set of registers and flags. If your subroutine changes these values, the original values are lost forever. If these values are important to your program, you need to save them before you change them and restore them back to their original values when the subroutine completes. Programmers typically do with the PUSH & POP instructions, which we'll discuss later. In addition, any well-written subroutine contains a header that describes what register and flag values the subroutine changes.

- The subroutine represents a different piece of code from the calling code. This being the case, the subroutine may inadvertently modify a register that the calling code is using. This is why all subroutine banners should contain a list of registers and flag values that the subroutine modifies. The subroutine in Figure 13-15 modifies two registers: r1 and r9; the two instructions also modify both the C & Z flags.

- We say that subroutines have "overhead". This means there are instructions associated with subroutines that we consider as doing "nothing useful". In this case, the CALL and RET instructions essentially do nothing except handle the administrative tasks of the program control associated with calling and returning from subroutines. The idea here is that since subroutines have this overhead, it is an ungood idea to have many short subroutines in your program unless you really have a good reason to do so.

```
(00)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)            AND    r0,r3         ; some random operation
(02)            CALL   My_sub        ; go off to do some special task
(03)            OR     r2,0x08       ; some other random operation
(04)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(05)
(06)   ;-------------------------------------------------------
(07)   ;- Subroutine: My_sub
(08)   ;- Description: This is an example showing the subroutine
(09)   ;-  calling/returning mechanism; it does nothing useful.
(10)   ;-
(11)   ;- Tweaked regs & flags: r1, r9, C, Z
(12)   ;-------------------------------------------------------
(13)   My_sub:   AND   r1,r2         ; some random operation
(14)             OR    r9,r3         ; more stuff
(15)             ...                 ; more stuff
(16)             RET                 ; return to calling program
(17)   ;-------------------------------------------------------
```

Figure 13-15: A program fragment showing typical subroutine invocation.

Table 13.8 provides more information regarding the CALL & RET instructions. The RTL column provides information regarding the underlying hardware. In this column, PC and SP are acronyms for "program counter" and "stack pointer", respectively. We'll provide more description of these values in the chapter dedicated to RAT MCU hardware.

Instruction	RTL	Example
CALL	PC ← imm_val (SP-1) ← PC SP←SP – 1	`CALL my_sub`
RET	PC ← (SP) SP ← SP + 1	`RET`

Table 13.8: Examples of the RAT MCU I/O instructions: CALL & RET.

Example 13.1

Write a subroutine that multiples a 4-bit number in register r8 with the 4-bit number in register r9. Place the result in register r10.

Solution: You'll quickly note that the RAT MCU instruction set does not include a multiply instruction. The solution therefore entails implementing multiplication by repeated addition algorithm. Figure 13-16 shows a possible solution to Example 13.1. The algorithm used in this solution takes advantage of the fact that multiplication is nothing more than repeated addition. Note that this solution contains a standard assembly language loop construct. In the code, one of the operands is decremented each time the code executes the ADD operation in the body of the loop. The body of the loop primarily "accumulates" the intermediate results of the add operations. In reality, this type of construct is used to implement a higher-level statement such as a *for* loop.

```
;-----------------------------------------------------------------------
;- subroutine: Mult_num  - places the effective multiplication
;-  of the number in register r8 with the number in register r9. It
;-  is expected that these number are limited to 4-bits or the
;-  result in register r10 may not be valid.
;-
;- Tweaked regs & flags: r9, r10, C, Z
;-----------------------------------------------------------------------
Mult_num:    MOV     r10,0x00    ; clear register r10
loop:        ADD     r10,r8      ; add multiplicand
             SUB     r9,0x01     ; decrement r9
             BRNE    loop        ; jump if r9 is not zero (add again)
             RET                 ; r9 is zero; result is in r10.
                                 ; return control to the main program
;-------------------------------------------------------------- ----------
```

Figure 13-16: Solution to the example: multiplication subroutine.

At first glance, the solution shown in Figure 13-16 appears to be adequate. However, upon further examination, you'll notice that there is a case where the solution fails. In particular, when the operand in the r9 register is a zero, the decrement after the first execution of the loop body will cause strange results. If you subtract one from zero in the RAT MCU architecture, the answer will be non-zero and strange things will happen. We refer to this as a boundary issue, where most programming errors tend to occur. We'll discuss these issues in more detail in a later chapter.

A better solution to this problem would be to first clear the accumulator register, and then check to see if either of the operands are zero. Figure 13-17 shows a better solution for this example. In this solution, if one of the operands is zero, the answer is known (it's zero) and control can return to the calling program (since the r10 register was previously loaded with zero). The solution in Figure 13-16 works in most cases, but it fails is one of the operands is zero.

```
;-----------------------------------------------------------------------
;- subroutine: Mult_num  - places the effective multiplication
;-  of the number in register r8 with the number in register r9. It
;-  is expected that these number are limited to 4-bits or the
;-  result in register r10 will not be valid.
;-
;- Tweaked regs & flags: r9, r10, C, Z
;-----------------------------------------------------------------------
Mult_num:    MOV     r10,0x00   ; clear register r10
             TEST    r8,0x00    ; set flags
             BRNE    ret1       ; jump to check other operand
             RET                ; return 0 if operand is 0
                                ;
ret1:        TEST    r9,0x00    ; set flags
             BRNE    loop       ; jump to do calculation
             RET                ; return 0 if operand is 0
                                ;
loop:        ADD     r10,r8     ; add multiplicand
             SUB     r9,0x01    ; decrement other operand
             BRNE    loop       ; jump if r9 is not zero (add again)
             RET                ; r9 is zero; result is in r10.
                                ; return control to the main program
;-----------------------------------------------------------------------
```

Figure 13-17: A better solution to the multiplication problem.

13.3.4.1 Special Subroutine Functionality

There are two other common functionalities associated with subroutines. First, in most viable programs, you'll find subroutines that call other subroutines. We refer to this functionality as *nested subroutine calls*, *nested*

calls or *nested subroutines*. Simple nested subroutine calls are common in assembly language programming both out of a sense of need and as a tool to keep programs neat and organized.

Another common issue associated with subroutines is when subroutines call themselves. When a subroutine calls itself, we refer to it as a *recursive subroutine call,* or simply *recursion*. We consider recursion a special type of nested subroutine. Recursion is a special animal in that it's somewhat hard to comprehend and is even trickier to actually use properly in a program. You can better understand recursion if you understand the underlying hardware implements subroutines (namely the CALL and RET instructions), but we're purposely excluding hardware details from this chapter. My personal thoughts about recursion are that you should avoid it at all costs. If you can't avoid it, make sure you understand it well enough to ensure it works properly in your code.

Remember that each CALL instruction causes the stack pointer to decrement one position. If you nest your subroutines calls too deeply, or if your recursion is too deep, the will be stack overflow and your program will die a violent death, probably without you knowing it. The next section discusses these issues in more detail.

13.3.4.2 Special Subroutine Issues

There are two issues that you need to be aware of regarding subroutines. First, there are generally limits to the levels of nesting you can have; these limits apply to both simple nested subroutines as well as recursion. Due to the nature of how the underlying hardware handles subroutine calls, each time you call a subroutine, the hardware writes to an address in memory. We refer to the area in memory that the hardware writes to as the *stack*. If you nest your subroutines too deeply, you stand a chance of running out of that underlying memory. We'll cover more details of the stack in later sections of this chapter.

The truth is that you never really run out of memory; what generally happens is that you start overwriting parts of memory that are critical to the proper operation of your program. In addition, the worst part is, it's really tough to figure out exactly why your program died as errors like this are somewhat intermittent (which are the hardest type of bug to fix). The last comment here is that the type of error you generate by nesting your subroutines too deeply is partially dependent on the exact architecture of the underlying hardware.

13.3.4.3 Stack Underflow and Overflow

The description of stack operations implemented with a memory device leaves open the option for improper stack operations. When you follow the proper stack protocols, you should never pop more items off the stack than you have previously pushed onto the stack. This sets up a condition of stack *underflow*. Likewise, you should never push more items onto the stack than you have memory locations in the stack. Remember, this stack is located in memory and the stack pointer is nothing more than a counter with increment and decrement inputs. We refer to this condition stack *overflow*.

Stack overflow and underflow conditions can cause disastrous effects on the operation of your program. It is possible to set up checks in software to ensure these conditions never happen but this requires processor time and code space for the instructions. The better solution is to write good code that naturally prevents such conditions from occurring.

13.3.4.4 General Rules of Proper Subroutine Usage:

Because there are not official constraints or rules regarding the use of subroutines, you should strive to follow a few basic guidelines when you use them.

- Subroutines should contain a piece of code that has some specific purpose. If each subroutine has a specific purpose, there is a greater chance you can reuse that code in another program. In addition, subroutines with specific purposes are easier to document and understand.

- All subroutines should be clearly delineated from other parts of the code by using an appropriate amount of comments. This promotes neatness and readability of your source code.

- It's generally a good idea to put all your subroutines at the end of your source code.

- All subroutines should contain a banner that provides the name of the subroutine, a description of what the subroutine does, a list of register arguments sent to the subroutine, and a list of what registers and flag that the subroutine modifies.

- Your subroutines should not be too short or too long. If you subroutines are too long, consider breaking them up into smaller subroutines and use nested subroutine calls. If your subroutines are too short, you stand the chance of having the overhead associated with your subroutine make your code inefficient in terms of running time (which is partially dependent on how often you call the subroutine).

13.3.5 Interrupts

The concept of interrupts is relatively simple. Essentially, an interrupt is a subroutine call that some device external to the MCU initiates. Recall that a normal subroutine call happens as a result of issuing a CALL instruction, thus is under internal control. There are generally three types of interrupts, which we somewhat describe below. For better or worse, the RAT MCU only has the capability of handling one external interrupt. Although this could be somewhat limiting, the operational characteristics of the RAT MCU's interrupt reflects the basic operation of external interrupts in modern microcontrollers.

1) External Interrupts: Some device external to the MCU generates this type of interrupt. We generally refer to these devices as peripherals and include such things as analog-to-digital converters, digital-to-analog converter, real-time clock (RTC) modules, and many other communication-type devices. The thing that makes these devices external is that they are physically connected to an interrupt pin on the MCU, which is a special pin in that it has the ability to generate interrupts in the MCU itself.

2) Internal Interrupts: Some device internal to the MCU generates this type of interrupts. We also refer to these devices peripherals and include the same devices as listed above. In other words, some MCUs contain these peripherals as part of the MCU itself in that these devices live on the interior of the IC. The RAT does not have any of these but just about every other microcontroller out there does. Generally speaking, once you start adding internal peripheral devices, you're necessarily dealing with a microcontroller as opposed to a microprocessor, as microprocessors are primarily CPUs with extremely limited memory and/or I/O capabilities.

3) Software-based Interrupts: We typically use these types of interrupts for debug functions and to promote genericity in the way the MCU handles interrupts. We don't generally see software-based interrupts often as the other two types of interrupts. This chapter does not discuss this type of interrupt, as the RAT MCU does not have that capability.

The basics of interrupts are that some device tells the CPU that it needs "service". In this context, service refers to notion that some devices need the MCU to stop executing the code it is currently executing, and to go execute some other special code. Having a device request service is a more efficient method than the CPU constantly asking the device if it needs attention (polling). The idea here is that we want the MCU to always be doing important and useful tasks; constantly asking a device if it needs service requires the MCU execute instructions. If the external device does not require service, the MCU has effectively wasted that processing

time. The interrupt model is inherently asynchronous in nature (not always though, but we can pretend it is for the sake of this discussion). This means that part of the mechanism that handles interrupts must deal with this asynchronous condition.

As mentioned earlier, many of the RAT MCU programs you've been writing thus far were using *polling* in order to "do something useful". In this context, the "something useful" statement refers to the fact that your program input some data from the outside world, performed some operations on that data, and then proceeded to output something related to the data to the outside world. Figure 13-18 shows the basic structure of how your programs up to now have looked up to now. This program consists of an endless loop that constantly monitors some inputs and writes to some output. Once again, we refer to this act of constantly checking the status of something as *polling*.

```
;-------------------------------------------------------------
;- Assembler Directives
;-------------------------------------------------------------
.CSEG                        ; code segment
.ORG  0x00                   ; start program somewhere
;-------------------------------------------------------------
                             ;
main:       OR    r0,00      ; do something
            IN    r1,0x22    ; so something
                             ; do a bunch more stuff
            OUT   r1,0x23    ; do even more stuff
            BRN   main       ; main loop
;-------------------------------------------------------------
```

Figure 13-18: The typical RAT program up until now.

This is a relatively simple approach to implementing a task and works quite well for simple programs such as the programs we've dealt with so far. The drawback of this approach is that it is an inefficient approach to performing a task. In official terms, the system *throughput*, or the total useful information processed or communicated during a specified time period is reduced because the MCU spends much of the processing effort querying an I/O device that, more likely than not, will not provide useful information most all of the time. A better approach would be to bypass the microcontroller constantly asking the I/O device whether it has new data and instead allow the I/O device to specifically tell the MCU that it needs to perform some action. We refer to the mechanism that implements this more efficient action as *interrupts*. We refer to programs using an interrupt approach to I/O as being *interrupt driven*.

The term *interrupt* comes from the fact the normal operation of the microcontroller is briefly *interrupted* to take care of some other special task. Once this task is *handled*, the microcontroller returns to the processing it was doing when the interrupt arrived. The basic model is that some peripheral device can *request service* from the MCU. We do this by allowing the external device to directly connect to MCU in a hardware sort of way. In the RAT MCU, we accomplish this by using a single external input, which we refer to as the *interrupt* input. The interrupt approach to I/O frees the MCU from the burden of polling and, in theory, allows the MCU to do other useful things and increase system throughput.

The concept of interrupts in the non-RAT environment is quite deep and it is common for microcontrollers to have complex interrupt architectures. Imagine for moment a complex digital system that contains many I/O devices. In such a system, polling each of these devices would usually be a bad option because it may take a long time for a given device to get the service it needs. The better option would be for all the devices to request service from the microcontroller only when they need it. The concepts of interrupt priority, interrupt latency, specialized interrupt hardware, and real-time concepts are important, but these are beyond the scope of this text. Keep in mind that even the most basic hardcore MCU has many interrupts. It's impossible to find an available MCU that has less than ten interrupts of various types, which underscores their importance in embedded systems programming.

Being that there is no one method used by all MCUs to handle interrupts, you'll soon discover that one of the first things you must do when working with a new MCU is to examine the interrupt architecture. First, you look at the architecture, then you look at the instruction set, then the I/O architecture, and finally, you look at the *interrupt architecture*. You'll need to establish the flavor and number of interrupts the microcontroller handles and how exactly the MCU handles the interrupts, since the use of polling rather than interrupts is an indication of a nooby programmer.

Interrupts are an extremely import part of any computer system. Thus, understanding is vital to writing good programs. Generally speaking, the notion of interrupts becomes more important with working with embedded systems and particularly at the assembly language level. In order to successfully work with interrupts, you must understand the low-level details of the interrupt architecture associated with the computer you're working with. The RAT MCU has about the simplest interrupt architecture there is, but despite this simplicity, it provides a complete overview of typical interrupt operations.

The notion of interrupts is relatively simple due to their similarity with subroutines. Stated as simply as possible, an interrupt is basically a subroutine call that is initiated by the hardware. In contrast, a subroutine is the program (software or firmware) initiates call. Figure 13-19 shows a high-level model of the RAT MCU hardware. Note that there is an input named "INT", which is shorthand notation for "interrupt". Speaking in high-level terms, when MCU detects the signal connected to the INT input at a '1' state, the RAT MCU executes a special subroutine. We usually refer to this special subroutine as the "interrupt service routine", or "ISR", but other people refer to it as the "interrupt handler" and other names that I can't think of now.

There are many advantages to using interrupts in embedded systems. Although they are something relatively new to programmers, you really must learn them in order to be a viable programmer. One thing to avoid is trying to avoid writing programs that use interrupts. The problem is that there are times when the programmer really should be using interrupts, and if the program does not use them, it's painfully obvious that the programmer is a novice programmer, which brings into question the quality of programming practices in the entire program.

When the RAT MCU detects that the INT input is asserted, the underlying hardware initiates a sequence of events. We won't discuss that sequence of events until we consider the hardware view of interrupts in another chapter, but it is something you should keep in mind as you read through this firmware/software version of interrupts. There are some other issues involved in the assertation of the INT signal, but we'll once again save those for the hardware chapter. For now, all you need to know is that there is some piece of hardware connected to the RAT MCU's INT input, which has the ability to generate an interrupt. Another way to look at it is that there is a device that physically connects to the RAT MCU that will somehow exert control over the executing RAT program.

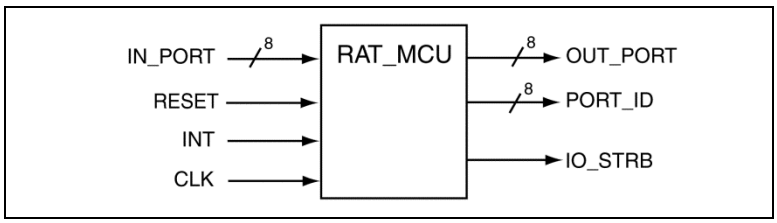


Figure 13-19: A high-level hardware view of the RAT MCU.

13.3.5.1 Enabling and Disabling Interrupts: SEI & CLI

Before you know anything about interrupts, you can easily understand some of the parameters that programmers can use to control interrupts. There are two RAT assembly language instructions that deal with enabling and disabling interrupts: the CLI and SEI instructions. Both of these instructions are of none-type instruction, as neither instruction requires operands. Table 13.9 shows examples of these two instructions. In

Table 13.9, the term “IF” stands for interrupt flag; the interrupt flag is a single bit in the RAT MCU hardware. When this bit is set, the interrupts are enabled; when this bit is cleared , the interrupts are disabled.

Instruction	RTL	Example
CLI	IF ← ‘0’	`CLI`
SEI	IF ← ‘1’	`SEI`

Table 13.9: Examples of the RAT MCU I/O instructions: CLI and SEI.

You’ve already seen the CLI instruction in several previous example problems. We use the CLI instruction to “prevent interrupts”; the instruction officially stands for “clear interrupt flag”, which we’ll discuss later. The SEI instruction officially “allows interrupts”; the instruction mnemonic officially stands for “set interrupt flag”. These terms are generally shorthand notion for what really happens in the hardware. When we say, “allows interrupts”, what we really mean is that we allow the MCU to process an interrupt that an external devices generates. When we "disable interrupts”, we don’t allow the MCU to process interrupts. In the latter case, the external device still can generate an interrupt, but the MCU simply ignores it, and continue with its normally scheduled processing. There is more to interrupts than this description, so we’ll discuss the full details in the hardware chapter.

There are two other issues associated with interrupts that the RAT assembly language programmer must be aware of in order to make the interrupt mechanism work properly. Here is a description of those issues.

1. In general, the hardware should disable the interrupts when the hardware powers up. Somewhere in the datasheet for the hardware you’re using should cover power-up issues such as initial values for sequential circuits. The RAT MCU powers-up with the interrupt disabled. Still, I prefer to have the first instruction in my assembly language program to be a CLI, just to “make sure” and allow me to sleep better at night.

2. The underlying RAT hardware does several things when it starts processing an interrupt. Once again, we’ll cover most of these items in the hardware chapter. However, what you need to know as a programmer is that the underlying hardware automatically disables the interrupts when the hardware acts on an interrupt. The reason for this is that it forces the programmer to officially enable the interrupts before another the RAT MCU can act on another interrupt. The idea here is that you generally don’t want to process the same interrupt more than once, which can happen if the hardware did not automatically disable the interrupts.

13.3.5.2 Saving the Context

The RAT MCU hardware automatically does what we refer to as saving the “state” of the MCU, or “current context”. The idea here is that the program is executing instructions and doing something important when it receives and acts on an interrupt. Acting on an interrupt makes the RAT MCU automatically execute the ISR. What you ideally want the MCU to do is stop the code that it is currently executing, execute the ISR code to completion, and then go back to the code that the MCU was executing when the MCU received the interrupt. The idea here is that if you can “save the state” of the MCU before you execute the ISR, then you can “restore” that state once the ISR completes execution and before you start executing the code you were executing when MCU received the interrupt.

Various MCUs out there have many different context saving mechanisms. The context saving mechanism in the RAT MCU simply copies the current values in the C and Z flags to the special places we refer to as the “shadow C” and “shadow Z” flags. We refer to these flags as the “shadow flags”. The shadow flags are a Hardwar concept that we’ll discuss further in the chapter that deals with underlying hardware issues in the

RAT MCU. Ideally, the context of the MCU should also include saving the registers, but the RAT MCU chooses not to consider them in the current context it automatically saves.

The notion of context saving is similar to protecting register values in subroutines. This is an important concept. As with standard subroutines, you must be careful to not allow the subroutine to alter the contents of registers that hold information that the call task considers important. In other words, since your ISR has no way of knowing what registers the MCU was using when it received an interrupt, you need to make sure that your ISR does not alter the context of any registers that the ISR uses.

13.3.5.3 Returning From ISRs: RETID & RETIE

We've already mentioned that ISRs are similar to subroutines. However, since the RAT MCU employs an automatic context saving mechanism and because the RAT MCU can enable and disable interrupts, we need special return-type instruction to use when we return from interrupts (which the program does when it completes processing of the interrupts). Table 13.10 shows examples of the two return from interrupt instructions. The two instructions are RETID, which returns from the ISR with the interrupts disabled (masked), and RETIE, which returns from the ISR with the interrupts enabled (unmasked).

The RETID and RETIE cause another function to occur in the underlying hardware. These two instructions officially "restore context". In the case of the RAT MCU, restoring context involves copying the values in the shadow flags to the regular flags. In other words, when the RAT MCU executes one of these instructions, the underlying hardware automatically copies the shadow flag values back to the regular flag values. Recall that acting on the interrupt causes the RAT MCU to automatically copy the normal flag values into the shadow flag values.

Table 13.10 provides more information regarding the RETID & RETIE instructions. The RTL column provides information regarding the underlying hardware. In this column, PC and SP are acronyms for "program counter" and "stack pointer", respectively. The terms "shadZ" & "shadC" stand for the shadow Z and C flags, respectively. We'll provide more description of these values in the chapter dedicated to RAT MCU hardware.

Instruction	RTL	Example
RETID	PC ← (SP) SP ← SP+1 Z← shadZ, C← shadC IF ← '0'	**`RETID`**
RETIE	PC ← (SP) SP ← SP+1 Z← shadZ, C← shadC IF ← '1'	**`RETIE`**

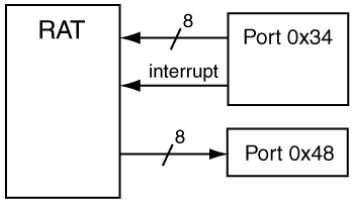
Table 13.10: Examples of the RAT MCU I/O instructions: RETID & RETIE.

13.3.5.4 Basic Interrupt Example Program

The previous sections presented a significant amount of information regarding interrupts. Let's now incorporate all this information into a simple "interrupt driven" program example. The following example does not really do anything useful, but it does show the programming mechanics of interrupts and how to use them.

Example 13.2

Consider a system where an external input device connects to the RAT MCU using Port Address 34. The device is also connects to the interrupt pin on the RAT MCU to allow it to generate an interrupt. When an interrupt occurs, read data from Port 34, compliment the data, and write the result to Port Address 48. The main loop of the program should not do anything except wait for the interrupt.



Solution: Figure 13-20 shows the solution to this example. There are several important issues in the solution program and we describe them all below. Keep in mind that this is a canned example and does not really do anything useful other than showing the major concepts involved in programming with interrupts. There is no proper order in which we can present this information; you'll need to read this multiple times in order to understand all the nuances involved in using interrupts.

- The code has a section for initialization; we mark this section with a comment and an "init" label. There are two steps in the initialization process. The first step is to clear the r31 register, which the code does using a reg/immed-type MOV instruction on line (10). The program officially uses the r31 register as a "flag" value. We use flags such as these often in assembly language programming; we typically use them to indicate something. As you'll see in this program, we use r31 as a flag to indicate an interrupt has occurred (this is an arbitrary register choice). The flag values are roughly speaking zero and non-zero in the register. The second thing the init code does is to enable the interrupts using the SEI instruction. When you are using interrupts, you want to keep the interrupts disabled until you complete any initialization code for the program; the notion here is that you want the program "set up" what it needs to before it receives the first interrupt.

- We refer to the section of code in lines (13-23) as the "main code" or sometimes as the *task code*. In essence, there are three sections of code in this program, 1) the initialization code, 2) the main code, and 3) the interrupt code. This is typical of well-structured assembly language programs, and of embedded systems programs in general.

- Lines (14-15) form what we refer to as the "main loop" of the task code. This is where the program spends most of it execution time. These two instructions form a tight loop that constantly checks the flag value in r31. So long as the flag value remains zero, the program stays in this loop. Note that the only way the value in r31 can possibly change is if the RAT MCU receives an interrupt. More on this in a later bullet.

- Line (33) shows the start of the interrupt service routine (ISR). The only purpose of the interrupt service routine is to set the r31 register to a non-zero value, which it does on line (33). The ISR is then complete, and program control returns to the task code after program executes the RETID instruction on line (34). We return with the interrupts disabled because we need to do some other processing (input, compliment, and then output) before we consider the interrupt as "handled"; we don't want another interrupt to occur before we can finish that processing.

- When the r31 register is non-zero, it means an interrupt has occurred and the ISR was executed. If you examine the code in this program, nothing can set the value in r31 other than receiving and acting on an interrupt (the MOV instruction in the ISR makes the value non-zero). In addition, you made sure the value in r31 was zero before you enabled the interrupts (you did this in the init code). Recall

that an interrupt is basically a subroutine initiated by hardware. The overall mechanism you should be seeing here is that an external device can have the ability to have the MCU implement a subroutine-like set of code we refer to as the ISR. When r31 is non-zero, program flow falls through the main loop to the other section of the main code.

- Once the program completes the desired processing in the main code, three events must occur. First, we must clear the r31 flag register, which we do on line (21) with a reg/immed-type MOV instruction. The idea here is that the flag should be set only once per interrupt. Second, we must re-enable the interrupts (we must unmask them). If we did not do this, we would not be able to act on any other interrupts. Thirdly, we need to branch back to the main loop of the main code. The process repeats itself *ad nauseum*.

- Lastly, and maybe most importantly, we need to place an unconditional branch instruction in the "interrupt vector address". The interrupt vector address is a pre-defined address that the program branches to when the RAT MCU executes an interrupt. The interrupt vector address for the RAT MCU is pre-set in the underlying hardware to 0x3FF, which is address of the last instruction in program memory. What we do is place an unconditional branch instruction at address 0x3FF so that when the RAT MCU acts on an interrupt, it correctly transfers program control to the ISR with the BRN instruction. In order to place the BRN instruction at 0x3FF, we use the ORG assembler directive with a 0x3FF argument. After the assembler directive and its 0x3FF argument, the assembler places the next instruction the program sees at address 0x3FF. Be careful not to place any more than one instruction after this directive, as 0x3FF is the last available instruction in program memory[16].

- Just admit it… the program is nicely formatted and is pleasing to the human eye.

[16] You actually can place instructions after the configuring the interrupt vector, but you first need to ORG the code back to a valid address in program memory.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .CSEG
(04)  .ORG  0x00                    ; memory location program
(05)  ;-------------------------------------------------------------------------
(06)  ;
(07)  ;-------------------------------------------------------------------------
(08)  ;- Initializations
(09)  ;-------------------------------------------------------------------------
(10)  init:       MOV     r31,0x00     ; initialize flag register r31
(11)              SEI                  ; unmask interrupts
(12)  ;-------------------------------------------------------------------------
(13)                                   ;
(14)  main:       CMP     r31,0x00     ; set flags
(15)              BREQ    main         ; branch if r31 = 0
(16)                                   ;
(17)              IN      r0,0x34      ; get data from port 0x34
(18)              EXOR    r0,0xFF      ; do something meaningful
(19)              OUT     r0,0x48      ; output data to port 0x48
(20)                                   ;
(21)              MOV     r31,0x00     ; clear interrupt flag
(22)              SEI                  ; enable the interrupts (unmask)
(23)              BRN     main         ; branch to main code
(24)
(25)  ;-------------------------------------------------------------------------
(26)  ;- The ISR:
(27)  ;- Description: This ISR sets bit0 in register r31 to act as flag
(28)  ;-  for the task code.
(29)  ;-
(30)  ;- Tweaked regs and flags: r31
(31)  ;-------------------------------------------------------------------------
(32)  ISR:                             ; (only a common label)
(33)              MOV      r31,0x01    ; set bit0 in r31
(34)              RETID                ; return with interrupt disabled
(35)  ;-------------------------------------------------------------------------
(36)
(37)  ;-------------------------------------------------------------------------
(38)  ;- Interrupt vector address
(39)  ;-------------------------------------------------------------------------
(40)  .ORG  0x3FF                      ; interrupt vector address
(41)              BRN     ISR          ; jump to interrupt service routine
(42)  ;-------------------------------------------------------------------------
```

Figure 13-20: The solution to Example 13.2.

13.4 The Scratch RAM

The RAT MCU contains a block of memory we refer to as the "scratch[17] RAM". We typically use the scratch RAM for many different items related to storing data. Since this block of memory is a RAM, we can both write to it and read from it. As you would imagine, there are RAT assembly language instruction that allow up to read from and write to scratch RAM. As with many MCUs, we refer to the action of reading from RAMs as a load operation, while we refer to writing to the RAM as a store operation. In regards to MCUs and computer architecture in general, we generally use load/store more often than we use read/write.

13.4.1 Accessing Scratch RAM: LD & ST

There are two types of instructions used to access the scratch RAM: load & store instructions. The RAT MCU instruction set uses the mnemonics of "LD" for load operations (reading from scratch RAM) and "ST" for store operations (writing to scratch RAM). In particular, these instructions transfer data from scratch RAM to a register (LD) or from a register to scratch RAM (ST). As with all memory, there are three items we need to

[17] The idea of "scratch" comes from scratch paper, which is paper you can do a lot of general stuff with.

tweak in order to user a memory: address data and control. With the RAT MCU, we provide the data and address as part of the instruction; the underlying hardware handles all the required control issues.

Additionally, there are two forms of load and store instructions; these forms differ by how they represent the address of the memory they access. Table 13.11 shows these instruction-types with associated code, RTL, and comments. The two types of instructions are reg/reg and reg/immed-type instructions. The reg/immed form of load instruction uses the immed value directly (the source operand) as an address into the scratch RAM. The reg/reg form of the LD and ST instructions use the contents of a register to supply the address into scratch RAM. We refer to the reg/reg form as the indirect form of the instruction because the instruction does not directly provide the address into scratch RAM as it does for the reg/immed form of the instructions. The address is instead contained in the register specified by the source operand of the reg/reg form of the instructions.

Instruction	Usage Example	RTL	Comment
LD (reg)	`LD r0,0x33`	Rd ← (immed)	Writes the value in register r0 into scratch RAM location 0x33
LD (indirect)	`LD r0,(r2)`	Rd ← (Rs)	Writes the value in register r0 into scratch RAM location specified by the value in register r2.
ST (reg)	`ST r0,0x33`	(immed) ← Rs	Reads the value in scratch RAM location 0x33 into register r0.
ST (indirect)	`ST r0,(r2)`	(Rd) ← Rs	Reads the value in the scratch RAM location specified by the value of register r2 into register r0.

Table 13.11: Examples and description of load (LD) and store (ST) instruction-types.

This indirect form of the LD and ST instructions allow for some useful programming constructs. In particular, the indirect form allows for efficient programming using “look-up tables”, or “LUTs”. We’ll examine a few examples of LUTs in a later chapter.

13.5 The Stack

The word stack has many different meanings for the people who use the word. Some of the definitions include haystack, smoke stack, pancake stack etc., but these are not the definitions we’ll be using here in technical-land. Here in technical-land, there are two main definitions of a stack. In software-land, the stack is one of the classic *abstract data types*, or, *ADTs*. In this context, the definition of an abstract data type is a data type that we describe in terms of the operations the data type supports rather than how we actually implement the data type. In other words, an ADT is a data type that is defined by its interface while placing no constraints on the implementation details. We’ll discuss the stack as an ADT in this chapter and examine the RAT MCU implementation details in a later chapter.

In the context of computer architecture, the stack has less of a computer science-type definition because we can describe the implementation of the stack in terms of simple hardware. Furthermore, the stack in standard computer architecture is an important part of any architecture because it is involved in several important flow control mechanisms; namely subroutines and interrupts.

The basic concept behind a stack is simple: it is nothing more than an object that stores data. The most basic definition of the stack lies in the description of the accessibility of the things that have been stored on the stack. The short definition of a stack is that the most recent thing that you place on the stack is the first thing that you can remove from the stack. We refer to this functionality as *Last In, First Out*, of *LIFO*.

Before going further, let's define a couple of terms for so we'll be speaking the same language regarding stacks. These terms are standard for any stack implementation; anyone dealing with stacks roughly knows what these terms mean in the context of how they are working with the stack. You need to know all of these terms for a hardware context, but you only need to know the first two terms if you're strictly a programmer. Note that none of these terms provides any pertinent implementation details.

- PUSH - This is the accepted term to mean that you are placing something onto the stack.
- POP - This is the accepted term meaning that you are removing something from the stack.
- Top of the Stack – We define the "top of the stack" to be the most recent object that we place, or *push* onto the stack. If the stack in empty (nothing has been pushed onto the stack), the top of the stack then has somewhat ambiguous meanings.
- Stack Pointer – We use the "stack pointer" as the "thing" we use to point to the top of the stack. If has placed objects onto the stack, then we base what the stack pointer is pointing at on the particular stack implementation. This will make more sense in the examples that follow.

There are two ways to demonstrate the operation of the stack. The first way is more of the computer science approach while the second way is more of a hardware approach. The hardware approach is more of what we're interested in as it is what the RAT MCU implement the stack in hardware, but the first approach makes for a nice introduction to the second approach. Figure 13-21 shows the first approach. Here is a description of the changes that take place in Figure 13-21; note that in Figure 13-21 we use the word "top" to indicate the top of the stack.

- Image 1: the stack in its empty state. For the empty stack, the top label is not well defined.
- Image 2: the stack after one item has been pushed onto the stack.
- Image 3: the stack after four items (three since image 2) have been pushed onto the stack. The number 34 was the first number pushed onto the stack, followed in order by 29, 19, and then 17.
- Image 4: the stack after one item has been popped from the stack (the number 17 was removed).
- Image 5: the stack after three items (two since image 4) have been removed from the stack.

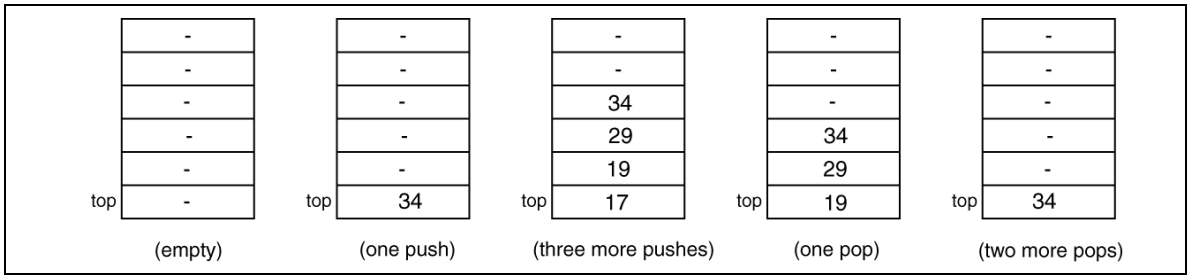


Figure 13-21: An example of a software-type description of a stack.

The key feature of the stack implementation shown in Figure 13-21 is that all the stack elements move each time a push or a pop operation executes. In this example, there is no real need to have or show an example of a stack pointer which is appropriate for the software description of a stack but would be inefficient for a hardware version of the stack because each stored stack item would need to be re-written for every push and pop operation. This being the case, we can better define the concept of a stack pointer by examining a hardware-type stack implementation. Figure 13-22 shows an example of a stack implemented in hardware, or more precisely, using a structured memory-type device.

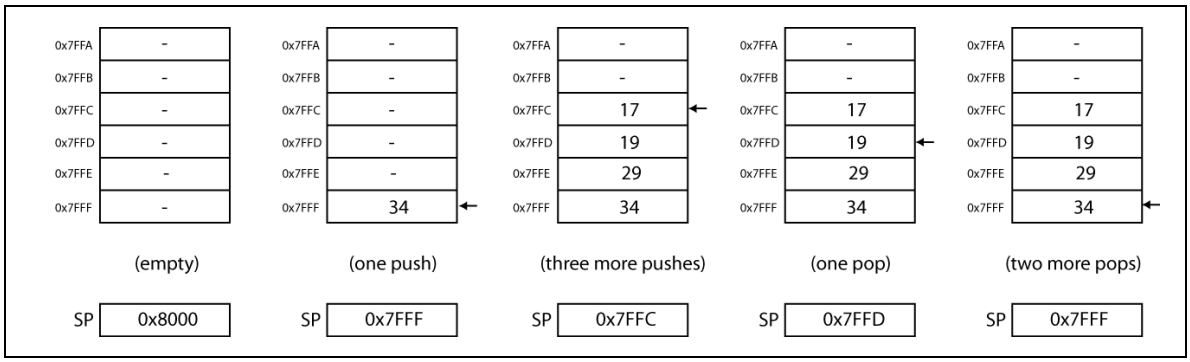


Figure 13-22: Implementation of a hardware-based (structured memory) stack.

In the example shown in Figure 13-22, we use a stack pointer to indicate the top of the stack. Brief descriptions of the changes that take place in this figure follow this paragraph. For this example, consider the stack location to start at 0x7FFF. We don't need to state the width or number of locations in the stack for this example.

- Image 1: the stack in its empty state. The stack pointer indicates the top of the stack (the box with the letters *SP* next to it). In its initial state, the stack pointer is officially not pointing to the memory associated with the stack. The stack pointer is defined at this point as indicated by the figure.
- Image 2: the stack after one item has been pushed onto the stack. A small arrow in addition to the stack pointer box is shown in the remaining figures to indicate the top of the stack.
- Image 3: the stack after four items (three since image 2) have been pushed onto the stack. The items on the stack were pushed in the following order: 34, 29, 19, and 17.
- Image 4: the stack after one item has been popped from the stack. Note that the item popped from the stack is not actually removed; the item is still there but the stack pointer is adjusted to point to a new top of the stack. If we were to push another item onto the stack after this point, it would necessarily overwrite the number 17.
- Image 5: the stack after three items (two since image 4) have been removed from the stack. Once again, we don't remove items from the stack, we simply adjust the stack pointer.

13.5.2 PUSH and POP Instructions

You could say that a set of instructions common to most MCUs is some form of a push and pop instruction. Not surprisingly, the RAT MCU does in fact have PUSH and POP instructions; both of these instructions are reg-type instructions. Similar to the LD and ST instructions, the PUSH and POP instructions provide a method of storing data. The main difference between PUSH/POP and LD/ST instructions is that the PUSH and POP instructions involve the stack and while LD and ST instructions do not. The mechanism associated with LD and ST instructions is rather simple because the instruction includes both the register and the scratch RAM memory address as part of the instruction. This allows the programmer to read from or write to any address in scratch memory.

The PUSH and POP instructions are different in that they only write to a certain address in memory. The address they write to is the address associated with[18] the top of the stack. In the underlying RAT MCU

[18] We use the words "associated with" because there are two main approaches to implementing a stack. Sometime the "top fo the stack" points at the last item placed on the stack and in other implementations, it points one address past the last item placed on the stack, which is the next unused address on the stack.

hardware, there is a register we use as the stack pointer; the value it holds an address we use to index into the stack. In short, LD and ST instructions are not associated with the stack while PUSH and POP instruction are.

Instruction	Usage Example	RTL	Comment
PUSH reg	**PUSH r0**	(SP-1) ← Rd, SP ← SP - 1	Writes the value in register r0 into scratch RAM location 0x33
POP REG	**POP r2**	Rd ← (SP), SP ← SP +1	Writes the value in register r0 into scratch RAM location specified by the value in register r2.

Table 13.12: Examples and description of PUSH and POP instructions.

The last issue to keep in mind as a RAT MCU programmer is that the stack in the RAT hardware is truly a LIFO device. This means that there is a special ordering you must apply when you use the PUSH and POP instructions. We can best show this with an example, such as the assembly code fragment in Figure 13-23. The example code in Figure 13-23 shows probably the most typical application of the PUSH and POP instruction. The idea here is that when you're going to "do something meaningful", you'll want to store the value in r10 and r11, with the thought that the meaningful thing you plan on doing will change the values and those register. Because you still need the values in r10 and r11, you store the values on the stack using two PUSH instructions. When you have complete your mission of "doing something useful", you restore the original values to r10 and r11 by executing two POP operations.

The most important point here is to realize that the PUSH and POP instructions store values on the stack and that the stack is a LIFO. This being the case, you need to pop items off the stack in the reverse order that you pushed them onto the stack. Note that in the example in Figure 13-23 that the POP instruction on line (11) undoes the PUSH instruction on line (04); similarly, the POP instruction on line (12) undoes the PUSH instruction on line (03).

```
(00)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)  ;- save r10 and r11 before you do something meaningful
(02)
(03)            PUSH   r10            ; save r10 on stack
(04)            PUSH   r11            ; save r11 on stack
(05)
(06)            ...                   ; do something meaningful
(07)
(08)  ;-------------------------------------------------------
(09)  ;- restore r10 and r11 after you do something meaningful
(10)
(11)            POP    r11            ; restore r10 value
(12)            POP    r10            ; restore r11 value
(13)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-23: A program fragment showing the proper order to use PUSH and POP instructions.

Figure 13-24 shows an example where the programmer mixed up the ordering of the POP instructions. In this case, if the programmer intended on restoring the original values to the register, then they failed. On the other hand, the code in Figure 13-24 does actually serve a useful purpose. In there was ever an occasion when you need to swap the values in two register, the out-of-order PUSH/POP pairs in Figure 13-24 do just that. This is sort of a well-known trick in assembly languages that takes advantage of the LIFO nature of the stack.

```
(00)   ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)   ;- save r10 and r11 before you do something meaningful
(02)
(03)            PUSH   r10            ; save r10 on stack
(04)            PUSH   r11            ; save r11 on stack
(05)
(06)
(07)            ...                   ; do something meaningful
(08)
(09)   ;-------------------------------------------------------
(10)   ;- restore r10 and r11 after you do something meaningful
(11)
(12)
(13)            POP    r10            ; restore r11 to r10
               POP    r11            ; restore r10 to r11
       ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 13-24: A program fragment showing the wrong order to use PUSH and POP instructions.

Figure 13-25 show an example that utilizes many of the concepts and instructions introduced in this chapter up to this point. The code in Figure 13-25 is a subroutine that examines ten scratch RAM memory locations starting at address 0xC0; the subroutine returns the number of locations that are zero. This subroutine uses an indirect memory reference instruction for this process. The use of indirect memory references is common in assembly language programming. The subroutine in Figure 13-25 is a great example of the versatility of indirect memory references; without such instructions, it would be impossible to use an iterative construct to facilitate such references. Here are some other interesting features of this subroutine.

- The general form of the code is: initialization, an iterative loop with a known loop count, an if/else construct embedded into the iterative construct, and a restoration of saved data. Note that this subroutine is a set of known actions and constructs.

- The subroutine lists what values the subroutines alters. Though the subroutine alters, r0, r1, and r2, it first saves those values before it alters them by pushing them onto the stack on lines (9-11). Saving values in this manner is part of the initialization phase of this subroutine. The subroutine later restores these values on lines(26-28). Note that the ordering on the PUSH & POP instructions supports the notion of the “last-in-first-out” stack.

- The code contains three working registers that serve to 1) act as an iterative count for the loop construct, 2) act as an index for the indirect memory references, 3) keep a binary count of the number of memory addresses that are zero. The subroutine stores values of these registers before it changes them. After saving these register values, the subroutine initializes them to 1) ten (it looks at ten value), 2) 0xC0 (the first address of the ten memory values), and 3) 0x00 (the binary count of zero values).

- Lines(23-24) handles the iterative loop maintenance issues. Line(22) is sort of loop maintenance, in that is increments the indirect memory reference register.

- Each time through the iterative loop, the subroutine examines a different memory address. It does this by using the indirect memory reference instruction on line(17). The subroutine uses the value in r1 as an address, or an index into scratch RAM. The subroutine loads the data from this address into a register.

- The code establishes the value read from scratch RAM using an CMP instruction on line(18). This configures the flags such that the following if/else construct works properly. If the value is zero, the code increments the count; otherwise it does nothing and continues on to the loop maintenance portion of the code.

```
(00)
(01)    ;-------------------------------------------------------------------------
(02)    ;- subroutine: Zero_count – This subroutine counts the number
(03)    ;   of memory starting at 0xC0 that are zero; this subroutines
(04)    ;   only considers a range of ten values. The subroutine returns
(05)    ;   the number in register r10.
(06)    ;-
(07)    ;- Affected regs & flags: r10, C, Z
(08)    ;-------------------------------------------------------------------------
(09)    Zero_count:
(10)    init:       PUSH    r0                  ; save state
(11)                PUSH    r1
(12)                PUSH    r2
(13)
(14)                MOV     r0,0x0A             ; set iterative count value
(15)                MOV     r1,0xC0             ; set index value
(16)                MOV     r2,0x00             ; clear tally
(17)    ;
(18)    loop:       LD      r10,(r1)            ; get data indirect from scratch
(19)                CMP     r10,0x00            ; set flags
(20)                BRNE    incr                ; jump over increment count
(21)                ADD     r2,0x01             ; increment tally count
(22)
(23)    incr:       ADD     r1,0x01             ; increment index
(24)                SUB     r0,0x01             ; decrement loop count
(25)                BRNE    loop                ; branch if more do to
(26)
(27)    restore:    POP     r2                  ; restore state
(28)                POP     r1                  ;
(29)                POP     r0                  ;
(30)
(31)                RET                         ; take it home jimmy
        ;-------------------------------------------------------------------------
```

Figure 13-25: Example using indirect memory access instruction.

13.6 Stack Initialization Instruction: WSP

As you know from previous sections, the stack is an important part of the RAT MCU architecture. The stack is important because it is the basic mechanism that allows PUSH & POP instructions to work properly and also provides a framework for subroutine calls and returns (CALL & RET) to function properly. These mechanism work because there is a stack pointer that is always "doing the right thing", which means that the data these mechanisms need to store is always done in such a way as to cause no harm to the executing program. In order to have the stack work properly, every program should set it to a known value in the initialization part of the program (before the main code). We don't include it in many of the program examples in this text in order to save space.

The WSP (write stack pointer) is a reg-type instruction copies a value from a register in the register file into the stack pointer. The stack pointer is an 8-bit value, which is the same width as the registers in the register file. When you use the WSP instruction, you must put a known value into the register that will become the new stack pointer prior to issuing the WSP instruction. Table 13.13 provides a description of the WSP instruction while Figure 13-26 shows an example of its usage in an actual program. The program in Figure 13-26 is similar to a previous program so we'll rely on the comments in the program for a description. The notion here is that line(13) moves a value into a register while line(14) copies that register value to the stack pointer, which effectively overwrites the value previously in the stack pointer.

In truth, the WSP instruction can be dangerous if you use it improperly. For example, if you were to rewrite the stack pointer in the middle of a program, you could "lose track" of data previously "pushed" onto the stack or you could also lose track of a return value for a subroutine call. Generally speaking, we only use the WPS instruction as part of the main program's initialization.

Instruction	Usage Example	RTL	Comment
WSP	**WSP Rd**	SP ← Rd	Copy the value from Rd into the stack pointer.

Table 13.13: Description of the WSP instruction.

```
(00)   ;--------------------------------------------------------------------
(01)   ;- Assembler Directives
(02)   ;--------------------------------------------------------------------
(03)   .EQU  DATA_IN_PORT       = 0xB6   ; example input port data
(04)   .EQU  DATA_OUT_PORT      = 0xB7   ; example output port data
(05)   ;
(06)   .CSEG
(07)   .ORG  0x10        ; memory location of instruction data
(08)   ;--------------------------------------------------------------------
(09)
(10)   ;--------------------------------------------------------------------
(11)   ;- Initializations
(12)   ;--------------------------------------------------------------------
(13)   init:    MOV     r0,0xFF            ; initialize stack pointer
(14)            WSP     r0                 ;  to 0xFF
(15)            CLI                        ; prevent interrupts
(16)   ;--------------------------------------------------------------------
(17)                                       ;
(18)   main:    IN       r1,DATA_IN_PORT   ; grab data, place in r0
(19)            EXOR     r1,0xFF           ; do something worthwhile
(20)            OUT      r1,DATA_OUT_PORT  ; output some data
(21)            BRN      main              ; go back to main loop
(22)   ;--------------------------------------------------------------------
```

Figure 13-26: A program the correctly initializes the stack pointer.

13.7 C Flag Manipulation Instructions: SEC, CLC

The C flag is one of two conditional flags on the RAT MCU. Recall that the number crunching instructions on the RAT MCU can assign the value of the condition flags based on the result of the number crunching operation. While the Z flag has one specific purpose for all instructions that are able to modify it, the C flag typically has more than one purpose. The multi-purposeness of the C flag allows the programmer to use it in many different and clever ways.

The big issue with the C flag is because of the shift instructions (LSL & LSR). Recall that executing these instructions causes the value stored in the C flag into the register associated with the instruction. This means that the programmer must be aware of the value of the C flag before they issues one of the shift instructions. The SEC & CLI instructions provide the programmer with the ability to put the C flag into a known state. Not surprisingly, viable RAT MCU assembly code often issues one of these instructions before executing a LSL or LSR instruction. Table 13.15 provides various levels of description regarding the SEC & CLI instructions.

Table 13.14

Instruction	Usage Example	RTL	Comment
SEC	**SEC**	C ← '1'	Assign '1' to value stored in the C flag
CLI	**CLI**	C ← '0'	Assign '0' to value stored in the C flag

Table 13.15: Description of the SEC & CLI instructions.

Figure 13-27 shows an example program where the SEC instruction is quite useful. Here is some useful explanation of the subroutine in Figure 13-27:

- The subroutine multiples a number by 16. Since there is no multiply instruction in the RAT MCU instruction set, we may think that we can simply add the number to itself 16 times. But the better solution is to utilize the fact that we need to multiply by a power of two; in this case a simple left shift is equivalent to a multiply by two. This means in order to multiply by 16, we need to perform four left shifts.

- Because we are multiplying by 16, we need to constrain the operator to be less than 16, which ensures the result will not overflow the 8-bit register width. Note that the subroutine does not explicitly check for this condition, which means the code relies on the programmer not misapplying the subroutine. Adding the check and generation an error condition would be a great idea. In addition, we could check the register value before the iterative portion of the code and return from the subroutine in cases where the value to be multiplied is zero.

- The subroutine works by iteratively shifting the value four times. The key here is to ensure that we always shift in a '0' to the working register, which we do by issuing a CLC instruction in the initialization portion of the program. Important to note where is that we only need to issue the CLC in the initialization portion of the subroutine. This is because if the value we need to multiply by 16 is truly less than 16, the four MSBs of the register will already be '0'. Recall for the LSL instruction, the MSB shifts into the C flag and the C flag shifts into the LSB.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Mult_16 – Multiples the value in r10 by 16. The result
(02)  ;   will only be correct if the value in r10 is less than 16.
(03)  ;-
(04)  ;- Affected Reg & flags: r10, C, Z
(05)  ;-------------------------------------------------------------------------
(06)  Mult_16:
(07)  init:       PUSH    r31         ; save r31 value
(08)              MOV     r31,0x04    ; iterative count
(09)              CLC                 ; clear the C flag
(10)
(11)  loop:       LSL     r31         ; shift MSB to C flag
(12)              SUB     r31,0x01    ; decrement loop count
(13)              BRNE    loop        ; branch if not done with loop
(14)
(15)  restore:    POP     r31         ; restore used registers
(16)              RET                 ; bring it on home
(17)  ;-------------------------------------------------------------------------
```

Figure 13-27: Subroutine that performs a multiply by 16.

13.8 Chapter Summary

- Program flow control refers to the order in which programs execute their instructions. Normal computer operations execute instructions sequentially, but computer programs must do something other than sequential execution if they are to do something useful.
- Assembly languages are said to have program flow control instructions, which are instruction that can cause program execution to go to some other place in the program other than the next instruction. These instructions are the conditional and unconditional branch instructions, subroutines, and interrupts.
- Interrupts provide a method for external hardware to execute a special subroutine typically referred to as the interrupt service routine (ISR). Much of the functionality associated with interrupts is the responsibility of the underlying hardware. Interrupt driven programs form the basis of embedded systems programming.
- The scratch RAM provides an area to store values. Values are stored in scratch RAM using the ST (store) instruction while values are read from scratch RAM using the LD (load) instruction.
- The stack is an abstract data type (ADT) that is used as to store values. The stack is implemented in hardware as a simple structured memory device with a stack pointer to act as the top of the stack. Data in the RAT MCU is saved to the stack using the PUSH instruction and removed from the stack using the POP instruction. The RAT stack is a last-in, first-out (LIFO) structure so programmers must be careful with the order in which they use the PUSH/POP instruction pair.
- PUSH/POP and CALL/RET instructions involve the operation of the stack. Since the stack is a physical device, there is a chance programmers can ruin the integrity of the stack via their use of PUSH/OP and CALL/RET instructions.

13.9 Chapter Exercises

1) In your own words, describe the what the term “program flow control” means.

2) Describe what exactly the following statement is referring too: “the Z flag always indicates the same characteristic for all instructions that tweak it; the characteristics indicated by the C flag are different depending on the executed instruction”.

3) Describe why a MOV instruction can be used as a true “nop”, while other instructions you know and love such as “`ADD r1,0x00`” are not true nops.

4) Describe the similarities between the CMP and TEST instructions.

5) Describe why you should never use a SUB instruction to establish a relationship between two numbers.

6) Described why you should never use an AND instruction to determine bit values of a given register.

7) Briefly describe why you think it is that every assembly language program has at least one unconditional branch instruction.

8) Briefly describe the difference between an iterative and condition loop construct.

9) Briefly describe why it is that you want to avoid iterative loops that only iterate once or twice.

10) Briefly describe whether a conditional loop or an iterative loop is more efficient in terms of loop overhead.

11) Describe a situation where a “lesser” amount of code requires more execution time than a “greater” amount of code. Assume these sets of code perform identical tasks.

12) Briefly describe why it is a good idea to provide a subroutine banner for all the subroutines in your program.

13) Briefly describe why it is a good idea to include a list registers modified by a subroutine in a subroutine banner.

14) What it the maximum depth you can nest subroutines with the RAT MCU?

15) Describe why there is a special need for instructions that only change the state of the C flag.

16) Describe where there is not an instruction that manipulates the Z flag in the same way that CLI & SEC manipulate the C flag.

17) Briefly describe why you feel it is that most MCUs don’t have bit-level instructions despite the fact that the MCU typically is operating on bits rather than bytes.

18) Both PUSH/POP instructions and LD/ST instructions allow the programmer to read and write data. Briefly describe the main differences between these types of instructions.

19) Briefly describe how the SUB/CMP instructions are similar to the AND/TEST instructions.

20) Describe why it is important to place a WSP instruction near the beginning of your code.

21) In order to be a good programmer, I swear I will always use a WSP instruction as the first executable instruction in my RAT MCU assembly language program.

22) In your own words, describe what we mean by “foreground” and “background” tasks in the context of RAT MCU assembly language programs.

23) The RAT MCU hardware takes steps to ensure that an ISR can’t be interrupted by another ISR. If you need to allow this to happen, describe how could you override the hardware in this case? This problem has nothing to do with RAT MCU hardware modifications.

14 Standard Assembly Language Programming Operations

14.1 Introduction

I consider assembly language programming an exercise in pulling "things" out of an assembly language "bag of tricks". The notion here is that there is generally speaking not that much you can do with assembly languages compared to higher-level languages. My feeling is that the most complicated part of learning to program in assembly languages is learning and keeping track of the various "tricks". These so-called tricks, are not really tricks; they're actually simple operations that would take you some extra time to be aware of if someone did not point them out to you. This chapter describes some of the more important considerations programmers should be aware of when writing robust assembly language code.

Main Chapter Topics

- **PASSING VALUES:** This chapter describes the concepts of passing values to and from subroutines.
- **ITERATIVE CONSTRUCT ISSUES:** This chapter describes some of the important underlying issues regarding iterative constructs.
- **MANIPULATING BITS:** This chapter describes the common bit manipulations found in assembly language programming.
- **COMPARING VALUES:** This chapter describes how the assembly language programmers establish equality between two values.
- **PROGRAMMING EFFICIENCY ISSUES:** This chapter provides an overview of concepts and terminology dealing with programming efficiency.
- **SEVEN-SEGMENT DISPLAY MULTIPLEXING:** This chapter provides several basic RAT assembly language example program

Why This Chapter is Important

This chapter is important because it describes some of the basic programming concepts and approaches beyond simple description of individual instructions.

14.2 Passing Values

The notion of "passing values" comes up quite often when programming computers. This is simply a matter of "what and how you send something" and "what and how that things sends something back". In higher-level languages, this primarily means the stuff you send to functions (formal parameters) and the stuff the function sends back (return values). For the RAT MCU, it simply means what data and how the calling code sends data to functions, and what data and how the subroutines returns data from the subroutine. As it turns out, this problem is not that complex with the RAT MCU; it's primarily a notion of learning the standard terminology when working the subroutines.

There are only three ways to pass data to subroutines in the RAT MCU: via registers, via scratch memory, or via the condition flags. Figure 14-1, Figure 14-2, and Figure 14-3 show examples of these three types of approaches programmers use to pass arguments to subroutines. Don't panic; although this appears to be a lot of code, the three subroutines are extremely similar. Each subroutine comprises of an init phase, an if/else statement, and a restore phase. The subroutine uses the if/else construct to either perform an addition or subtraction operation. A "passed" value decides on which operation to perform; the three subroutines differ in how the calling code passes values to the subroutines. Here are some more exciting details.

The subroutine in Figure 14-1 uses register r0 to pass a value to the subroutine. The if/else construct depends on the value in r0. Line(10) uses a CMP instruction to update the value in the Z flag, which enables the if/else clause to work properly. In the end, the calling code passes information of this subroutine via r0, r1, and r2. The subroutine passes data back to the calling code via r3, C and Z.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Add_Sub – Adds (r1 + r2) or subtracts (r1-r2) as
(02)  ;   determined by the value in r0. If r0=0, then add; otherwise
(03)  ;   Subtract. Result is returned in r3
(04)  ;-
(05)  ;- Affected regs & flags: r3, C, Z
(06)  ;-------------------------------------------------------------------------
(07)  Add_sub:
(08)  init:       PUSH    r1          ; save value
(09)
(10)  if:         CMP     r0,0x00     ; check r0 for required operation
(11)              BREQ    add_me
(12)
(13)  sub_me:     SUB     r1,r2       ; subtract operation
(14)              BRN     done        ; jump to closing stuff
(15)
(16)  add_me:     ADD     r1,r2       ; subtract operation
(17)
(18)  done:       MOV     r3,r1       ; copy result to r3
(19)
(20)  restore:    POP     r1          ; restore used registers
(21)              RET                 ; bring it on home
(22)  ;-------------------------------------------------------------------------
```

Figure 14-1: Passing data to subroutine via register.

The subroutine in Figure 14-2 uses the C flag to pass the value to the subroutine. The if/else construct depends on the C flag and directly uses a branch instruction (BRCC) on line(10). In the end, the calling code passes information to the subroutine via r1, r2, and the C flag. The subroutine passes data back to the calling code via r3, C and Z. Note that the code changes the original passed value of C based on the arithmetic operation in the subroutine.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Add_Sub – Adds (r1 + r2) or subtracts (r1-r2) as
(02)  ;   determined by the value in the C flag. If C=0, then add; otherwise
(03)  ;   Subtract. Result is returned in r3
(04)  ;-
(05)  ;- Affected regs & flags: r3, C, Z
(06)  ;-------------------------------------------------------------------------
(07)  Add_sub:
(08)  init:        PUSH    r1          ; save value
(09)
(10)  if:          BRCC    add_me      ; check C flags
(11)
(12)  sub_me:      SUB     r1,r2       ; subtract operation
(13)               BRN     done        ; jump to closing stuff
(14)
(15)  add_me:      ADD     r1,r2       ; subtract operation
(16)
(17)  done:        MOV     r3,r1       ; copy result to r3
(18)
(19)  restore:     POP     r1          ; restore used registers
(20)               RET                 ; bring it on home
(21)  ;-------------------------------------------------------------------------
```

Figure 14-2: Passing data to subroutine via flag.

The subroutine in Figure 14-3 uses the data in a memory location in order to pass a value to the subroutine. The subroutine then reads that value from scratch memory, and set the condition flags using the CMP instruction on line(11). The if/else construct depends on the Z flag and uses a conditional branch instruction (BREQ) on line(12). In the end, the calling code passes information to the subroutine via r1, r2, and the value in scratch memory address 0x47. The subroutine passes data back to the calling code via r3, C and Z.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Add_Sub – Adds (r1 + r2) or subtracts (r1 - r2) as
(02)  ;   determined by the value in data memory location ox47. The
(03)  ;   result is returned to the calling code in r3.
(04)  ;-
(05)  ;- Affected regs & flags: r3, C, Z
(06)  ;-------------------------------------------------------------------------
(07)  Add_sub:
(08)  init:        PUSH    r1          ; save value
(09)
(10)               LD      r3,0x47     ; get value from scratch RAM
(11)  if:          CMP     r3,0x00     ; check r0 for required operation
(12)               BREQ    add_me
(13)
(14)  sub_me:      SUB     r1,r2       ; subtract operation
(15)               BRN     done        ; jump to closing stuff
(16)
(17)  add_me:      ADD     r1,r2       ; subtract operation
(18)
(19)  done:        MOV     r3,r1       ; copy result to r3
(20)
(21)  restore:     POP     r1          ; restore used registers
(22)               RET                 ; bring it on home
(23)  ;-------------------------------------------------------------------------
```

Figure 14-3: Passing data to subroutine via scratch memory.

The subroutine in Figure 14-4 uses register r0 to pass a value to the subroutine. The if/else construct depends on the value in r0 and updates the condition flags using the CMP instruction on line(10). The if/else construct depends on the Z flag and uses a conditional branch instruction (BREQ) on line(11).The subroutine in uses the data in a memory location 0xAA in order to pass a value from the subroutine to the calling code. In the end, the calling code passes information to the subroutine via r0, r1, and r2; the subroutine passes data back to the calling code via memory location 0xAA.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Add_Sub – Adds (r1 + r2) or subtracts (r1-r2) as
(02)  ;   determined by the value in r0. If the r0=0, then add;
(03)  ;   otherwise subtract. Result is returned in memory address 0xAA.
(04)  ;-
(05)  ;- Affected regs & flags: C, Z
(06)  ;-------------------------------------------------------------------------
(07)  Add_sub:
(08)  init:        PUSH    r1          ; save value
(09)
(10)  if:          CMP     r0,0x00     ; check r0 for required operation
(11)               BREQ    add_me
(12)
(13)  sub_me:      SUB     r1,r2       ; subtract operation
(14)               BRN     done        ; jump to closing stuff
(15)
(16)  add_me:      ADD     r1,r2       ; subtract operation
(17)
(18)  done:        ST      r1,0xAA     ; copy result to mem address
(19)
(20)  restore:     POP     r1          ; restore used registers
(21)               RET                 ; bring it on home
(22)  ;-------------------------------------------------------------------------
```

Figure 14-4: Passing data from the subroutine via scratch memory.

14.3 Iterative Construct Issues

Assembly languages generally support two types of iterative constructs: 1) when we know in advance how many times we iterate, and 2) when we don't know in advance how many times we need to iterate. Coding iterative constructs is mostly a cookbook operation, and 90% of the time you'll be able to code a great iterative construct in your sleep (as your instructor lectures). Then again, 90% of the errors you make in any given assembly language program will be associated with a loop construct. Because of this, you need to be the most careful when you're coding your constructs. This section examines a few areas that cause problems for even the most experienced programmers.

14.3.1 Do-While vs. While Iterative Constructs

There are two main forms of looping constructs in assembly language program. Recall that the general form of a loop construct is that you need to perform some set of instructions a given amount of times; you may not know in advance how many times you need to iterate. When the iteration value is based on a condition, you may not know the condition in advance. This leads to a potential issue with the general form of loop constructs.

The two types of iterative constructs are 1) the do-while loop, and, 2) the while loop. These two loops have one simple difference: the do-while loop guarantees that the loop iterates at least one time regardless of the loop count, which means the MCU executes the body of the loop construct at least one time. The while loop may not iterate at all based on the loop count. This creates a problem when the loop count is variable based on some parameter in the overall program. The exact problem is that you can't use a do-while loop in cases where the loop count may be zero. This is a problem because the do-while loop by definition always iterates at least once.

Figure 14-5 show a fragment of code showing a do-while loop with an error. The code inputs value using an IN instruction and then uses that value as an iterative count. This code works perfect, except for one case: when the input value is zero, the body of the iterative loop should not execute. In the case where the input value is zero, the loop executes, then decrements the register storing the loop count (r0). When the code decrements zero, the result is 0xFF. Thus, the loop will execute 256 times when it should not have executed a

single time. This is a common error in assembly language programming; always be aware of it when you code up your loops.

```
(00)
(01)
(02)    ;-----------------------------------------------------------------
(03)    ;- Assembler Directives
(04)    ;-----------------------------------------------------------------
(05)    .EQU  DATA_IN_1          = 0x55   ; input port
(06)    .EQU  DATA_IN_2          = 0x56   ; input port
(07)    .EQU  DATA_OUT           = 0x57   ; output port
(08)    ;
(09)    ;-----------------------------------------------------------------
(10)    ; do-while iterative loop construct without "protection"
(11)    ;-----------------------------------------------------------------
(12)
(13)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(14)    ;
(15)                IN      r0,DATA_IN_1       ; load iterative count value
(16)    ;
(17)    loop:       IN      r2,DATA_IN_2        ; get some data
(18)                EXOR    r2,0xFF             ; toggle input data
(19)                OUT     r2,DATA_OUT         ; output some data
(20)                SUB     r0,0x01             ; decrement iteration variable
(21)                BRNE    loop                ; do again if count is non-zero
(22)                                            ;
        loop_done:                              ; do something else…
        ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-5: Example of a do-while with an potential error.

Figure 14-6 shows a fix for the issue in Figure 14-5. When the input value has the possibility of being zero, the code should check that value first. If the value is zero, program control should pass over the iterative loop. When the input value is non-zero, the loop executes normally. The addition of these two instructions will definitely save your arse in some situations, which of course justifies including the extra code.

It's probably worth nothing here that the code in Figure 14-6 is probably closer to being a while loop than it is a do-while loop. To be complete, the code in Figure 14-7 is a proper while loop construct; this code is functionally equivalent to the code in Figure 14-6. You could probably argue that one set of code is more efficient, but let's not go there now.

```
(00)    ;-----------------------------------------------------------------
(01)    ;- Assembler Directives
(02)    ;-----------------------------------------------------------------
(03)    .EQU  DATA_IN_1          = 0x55   ; input port
(04)    .EQU  DATA_IN_2          = 0x56   ; input port
(05)    .EQU  DATA_OUT           = 0x57   ; output port
(06)    ;
(07)    ;-----------------------------------------------------------------
(08)    ; do-while iterative loop construct with "protection"
(09)    ;-----------------------------------------------------------------
(10)
(11)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(12)    ;
(13)                IN      r0,DATA_IN_1        ; load iterative count value
(14)                CMP     r0,0x00             ; see if count is zero
(15)                BREQ    loop_done           ; jump over loop if zero
(16)    ;
(17)    loop:       IN      r2,DATA_IN_2        ; get some data
(18)                EXOR    r2,0xFF             ; toggle input data
(19)                OUT     r2,DATA_OUT         ; output some data
(20)                SUB     r0,0x01             ; decrement iteration variable
(21)                BRNE    loop                ; do it again if count non-zero
(22)                                            ;
(23)    loop_done:                              ; do something else…
(24)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-6: Example of do-while construct that that will always work properly.

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-----------------------------------------------------------------------
(03)  .EQU  DATA_IN_1          = 0x55   ; input port
(04)  .EQU  DATA_IN_2          = 0x56   ; input port
(05)  .EQU  DATA_OUT           = 0x57   ; output port
(06)  ;-----------------------------------------------------------------------
(07)  ; do-while iterative loop construct with "protection"
(08)  ;-----------------------------------------------------------------------
(09)
(10)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(11)  ;
(12)              IN      r0,DATA_IN_1        ; load iterative count value
(13)              CMP     r0,0x00             ; set flags
(14)
(15)  loop:       BREQ    loop_done           ; jump over loop if zero
(16)              IN      r2,DATA_IN_2        ; get some data
(17)              EXOR    r2,0xFF             ; toggle input data
(18)              OUT     r2,DATA_OUT         ; output some data
(19)              SUB     r0,0x01             ; decrement iteration variable
(20)              BRN     loop                ; do it again if count non-zero
(21)                                          ;
(22)  loop_done:                              ; do something else…
(23)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-7: Example of while loop construct.

14.3.2 Off-By-One Issues

Off-by-one issues essentially means that you mean think your code executes a loop X number of times, but the loop is actually executing either X+1 or X-1 number of times. Off-by-one errors are particularly common in assembly language programming because coding at such low levels forces you to have a 100% understanding of how the instruction actually work[19]. Once again, some loops are easy: you've coded a million of them, you never make a mistake. Coding these loops is easy because understanding the loop parameters are not too complicated. But when you need to code a loop construct that does something sort of special, you're more apt to make a mistake[20].

Figure 14-8 and Figure 14-9 show examples of a potential problem area. The code in Figure 14-8 shows a typical do-while loop that executes the body of the loop eight times. This loop is most likely the type of iterative construct you be using most often in assembly language programming. The code in Figure 14-8 works properly. The code in Figure 14-9 is similar to the code in Figure 14-8; there is only one difference. The code in Figure 14-9 bases the test of the iterative count on the C flag, as opposed to the Z flag, as is done in Figure 14-8. Keeping the count at eight and testing the C flag will cause the loop to iterate nine times rather than the desired eight times, which is because the one needs to be subtracted from zero in order for the SUB instruction so set the C flag. You can still use the loop in Figure 14-9, but you would need to decrement the loop count before you started executing the loop.

In real life, it make no sense to "decrement the loop count" before you started executing the loop. While these two examples are canned, there are instances in assembly language programming when testing the C flag makes more sense than testing the Z flag. Often times in computerland, you must examine something that is "zero-based", and you're using the iterative count as an index. For example, you want to examine the first ten locations in scratch RAM. The first ten locations in a RAM start at address zero and continue to address nine (0-9). In this case, starting your loop count at nine and using the C flag to test when the count transitions from 0 is a better solution. Keep this idea in mind; it really comes in handy.

[19] Which is not as true for higher-level languages (or at least it requires a different sort of understanding).

[20] In truth, coding loop constructs become so second nature, that you forget why it is the loop you coded actually works properly. Then when you have a special loop to code, meaning a loop that is not as straight-forward as all the other loops you coded, you have to really be careful because you've forgotten how the loops actually work. Welcome to assembly language programming.

```
(00)
(01)    ;-------------------------------------------------------------------
(02)    ; typical loop (do-while)
(03)    ;-------------------------------------------------------------------
(04)
(05)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(06)                MOV     r0,0x08             ; set iterative count value
(07)    ;
(08)    loop:       IN      r2,DATA_IN_2        ; get some data
(09)                EXOR    r2,0xFF             ; toggle input data
(10)                OUT     r2,DATA_OUT         ; output some data
(11)                SUB     r0,0x01             ; decrement iteration variable
(12)                BRNE    loop                ; do again if count is non-zero
(13)                                            ;
        ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-8: Typical do-while loop construct.

```
(00)
(01)    ;-------------------------------------------------------------------
(02)    ; common loop error (do-while) w
(03)    ;-------------------------------------------------------------------
(04)
(05)    ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(06)                MOV     r0,0x08             ; set iterative count value
(07)    ;
(08)    loop:       IN      r2,DATA_IN_2        ; get some data
(09)                EXOR    r2,0xFF             ; toggle input data
(10)                OUT     r2,DATA_OUT         ; output some data
(11)                SUB     r0,0x01             ; decrement iteration variable
(12)                BRCC    loop                ; do again if C flag set
(13)                                            ;
        ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-9: Example of a do-while with an potential error.

14.4 Bit Manipulations for MCUs

MCUs, or microcontrollers as some people call them, were designed to do exactly what their name implies: control things. By things, we mean external computer peripherals: MCUs monitor status inputs and send out control outputs. Recall that this functionality is similar to FSMs, but MCUs control things via software (or firmware) while FSM are purely a hardware-oriented device. The way MCUs control things at a low level is by the manipulating bits; thus, bit manipulations are a key element in writing meaningful programs for MCU. This section describes some of the details regarding the finer points of bit manipulations.

14.4.1 Tweaking Bits

Bit-tweaking is a well-known assembly language trick. In this context, we use the word “tweaking” to mean modifying bit values. There are only four things you can do with a bit: 1) setting, 2) clearing, 3) toggling (complimenting), and, 4) holding the bit value (doing nothing). The notion of tweaking bits is slightly misleading because MCUs such as the RAT MCU have instructions that only operate on complete register values. More specifically, the three logic-type instructions (AND, OR, and EXOR) operate on the entire register, which is why we refer to them as bit-wise operations. Although many modern MCUs have instruction that can perform logic-type operations on individual bits of a register, most do not.

Having instructions that manipulate individual bits are handy but they make the instruction set larger than it needs to be, which is why most instruction sets don’t have instructions that act on individual bits. One of the unstated requirements of assembly language programming is that you need to be clever. In other words, you need to work with the instructions you have to do what you need to do with a reasonable amount of complexity. You won’t always have the exact instruction you need every time but the instruction set should

always have the functionality to eventually create the operation you looking for (although it may take a few instructions instead of just one instruction).

Table 14.1 shows the standard assembly language programmers use to set, toggle, clear, and hold bits with bits using an assembly language. There are other ways to perform bit manipulations, but these are the accepted way in assembly language. When you use these methods, everyone knows what you're doing without putting too much thought into it. If you don't use these approaches, you'll have people wondering what you're doing, and thinking that you're an ungood programmer. There are many ways to "hold" bits; Table 14.1 shows three of the more common approaches. We'll use this information in the following section.

Bit Operation	How to do it
setting	Logical OR with '1'
toggling	Logical XOR with '1'
clearing	Logical AND with '0'
holding (do nothing)	Logical OR with '0' Logical XOR with 0 Logical AND with '1'

Table 14.1: Assembly language tricks to die for.

14.4.2 Bit Masking

Since we generally use MCU to control various computer peripheral devices, it would make sense that we can use single bits to control these devices rather than entire bytes. The issue with most MCUs is that they can only operate on eight bits at a time (one byte); the RAT MCU is one of these devices The result is that your assembly programs typically require the use of *bit-masks* in order to manipulate individual register bits. The bit-masks, combined with the monitoring of the condition flags, allows the microcontroller to perform different functions based on the status of individual bits of registers rather than the entire register. As you can probably imagine, bit-masking is really useful and common in assembly languages. Table 14.1 shows a few examples regarding bit-mask possibilities.

There are two main uses for bit masks: 1) checking individual input bits in a register, and, 2) assigning values to individual bits in a register. In most cases, your MCU is controlling some peripheral device, which means your MCU is typically reading status inputs from external devices and assigning control outputs to external devices. For the RAT MCU, we input values (IN instruction) from the outside world into a register, where we can then "check" them; we then output values (OUT instruction) to the outside world from a register. Keep these issues in mind when you browse these examples.

Example		Explanation
`OR`	`r1,0x02`	`Sets bit-1 in r1 (no other bits change)`
`OR`	`r8,0x0F`	`Sets bits (0-3) in r8 (no other bits change)`
`AND`	`r9,0x7F`	`Clears bit-8 in r9 (no other bits change)`
`AND`	`r17,0x0C`	`Clears bits (2-3) r17 (no other bits change)`
`EXOR`	`r22,0xEE`	`Toggles all bit except bits (4,0) in r22 (no other bits change)`
`EXOR`	`r21,0x81`	`Toggles bits (7,0) in r21 (no other bits change)`

Table 14.2: A few example bit-masks.

Figure 14-10 shows a fragment of code that utilized bit-masking. Here are the important points regarding the code in Figure 14-10.

- The idea behind the program fragment is that it is monitoring the switches; you can assume the switches are associated with the development board. The program provides a link the input ports as assembler directives. The program fragment is only concerned about the value of bit-3, which we consider the fourth bit from the right side of the register. When this bit-3 is set, the program does something; if the bit is not set, the program continues to monitor the input port.

- As with well-written assembly language programs, this program places all of the .EQU assembler directives at the beginning of the program. There are two directives to define the input ports; but we're interested in the directive used to define the bit mask. Line (06) show defines a bit-mask as 0x08, which means all the bit values are '0' except for the third one from the right.

- The code fragment utilizes the bit-mask on line (22) using an AND instruction. The AND instruction ANDs the value input to the value from the previous IN instruction with the bit-mask. Since seven bits of the bit-mask value are already '0', the result of the AND instruction will either be 0x00 if the bit-3 is cleared or 0x80 if bit-3 is set. The key here is that the AND operation set the value of the Z flag based on the result of the AND operation. In this way, if bit-3 is set, then the switch is on and Z='0'; otherwise if bit-3 is cleared, then the switch is off and Z='1'.

- Because the AND operation set the Z flag based on the value of bit-3 exclusively, if bit-3 was set, program control will drop through the BRNE statement on line (23) and go on to do other things (which this code fragment does not list). If bit-3 was not set, then the program will take the branch associated with the BRNE instruction.

- A side effect of the AND operation is that seven of the eight bits in register r0 may change. Because seven of the eight bits are being logically ANDed with '0', they will be set to '0' if they are not already '0'. Losing the values of the input switch can be problematic.

```
(00)  ;---------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;---------------------------------------------------------------------
(03)  .EQU SWITCHES = 0x30    ; switches address = port 0x30 (input)
(04)  .EQU LEDS     = 0x0C    ; LED address = port 0x0C (output)
(05)  ;
(06)  .EQU SW3_MASK = 0x08    ; mask bit 3 of 8-bit value ("00001000");
(07)  ;---------------------------------------------------------------------
(08)  ;
(09)  ;---------------------------------------------------------------------
(10)  .CSEG                   ; indicates code segment
(11)  .ORG   0x00             ; sets the code segment counter to 0x00
(12)  ;---------------------------------------------------------------------
(13)  ;
(14)  ;---------------------------------------------------------------------
(15)  ; The code listed below assumes that the button 7 is the far left of
(16)  ; eight buttons; button 0 is the right-most button. An un-pressed
(17)  ; button is considered to be in a zero state.
(18)  ;---------------------------------------------------------------------
(19)  init:   CLI                 ; prevent interrupts
(20)          ;
(21)  main:   IN      r0, SWITCHES   ; grab switch status
(22)          AND     r0, SW3_MASK   ; clear all but button 3 (mask button3)
(23)          BRNE    main           ; jump if not pressed
(24)          ; ...
(25)  thing:  ; Some other code here
(26)          ;  that does something useful
(27)          ; ...
(28)          BRN     main           ; jump back to main loop
(29)  ;---------------------------------------------------------------------
```

Figure 14-10: A code fragment showing a typical bit-masking operation.

14.4.3 The TEST Instruction

Line (22) in Figure 14-10 is a bit-masking operation that has the ability to alter program flow control based on the value of a single bit in a register. As you'll be seeing, this bit-masking and bit-testing is a particularly common operating in assembly languages, particularly with MCUs that we use in control applications. The notion here is that the individual bits are associated with either control or status of some external item. The problem is the AND instruction on line (22) is that it is only checking the status of a single bit, yet it is altering the values of all the other bits in the register. This situation is non-optimal.

The solution in this case is to use the TEST instruction instead of the AND instruction. The TEST instruction is the same as the AND instruction in that it performs and AND operation and sets the Z flag according to the result of the operation. However, it differs from the AND operation in that the RAT MCU does not write the result to the destination register. What the TEST instruction provides is a way to check individual values in registers without altering any data in the register. The TEST instruction is thus a very powerful instruction and you'll use it quite often in your assembly language programming career (even if it is a short one). Table 14.3 shows information regarding the reg/reg and reg/immed forms of the TEST instruction. Figure 14-11 shows the program in Figure 14-10 written using a TEST instruction in place of the AND instruction.

It is obvious that you have the ability to use either a AND or TEST instruction in the bit-masking operation. The question is when should you use one over the other? The answer is simple. If you're truly performing an AND operation and you need the result of the AND operation, than you must use the AND instruction. On the other hand, if you're checking the status of the bits in a register, then you should use the TEST instruction. Using the TEST instruction for bit-masking operations is a form of self-commenting in that if someone sees the TEST instruction, they immediately assume you're testing something. If you use the AND instruction instead of the TEST operation, people assume you'll need the result of the AND operation and they'll be confused when you don't actually use the result in your program. This is similar to the differences between the SUB and CMP instructions we discussed previously.

Instruction	Usage Example	RTL	Comment
TEST reg,reg	`TEST r1,r2`	Rd · Rs	Writes the value in register r0 into scratch RAM location 0x33
TEST reg,immed	`TEST r1,0x0F`	Rd · immed	Writes the value in register r0 into scratch RAM location specified by the value in register r2.

Table 14.3: Examples and description of PUSH and POP instructions.

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-----------------------------------------------------------------------
(03)  .EQU SWITCHES = 0x30    ; switches address = port 0x30 (input)
(04)  .EQU LEDS     = 0x0C    ; LED address = port 0x0C (output)
(05)  ;
(06)  .EQU SW3_MASK = 0x08    ; mask bit 3 of 8-bit value ("00001000");
(07)  ;-----------------------------------------------------------------------
(08)  ;
(09)  ;-----------------------------------------------------------------------
(10)  .CSEG                   ; indicates code segment
(11)  .ORG   0x00             ; sets the code segment counter to 0x00
(12)  ;-----------------------------------------------------------------------
(13)  ;
(14)  ;-----------------------------------------------------------------------
(15)  ; The code listed below assumes that the button 7 is the far left of
(16)  ; eight buttons; button 0 is the right-most button. An un-pressed
(17)  ; button is considered to be in a zero state.
(18)  ;-----------------------------------------------------------------------
(19)  init:   CLI             ; prevent interrupts
(20)          ;
(21)  main:   IN      r0, SWITCHES   ; grab switch status
(22)          TEST    r0, SW3_MASK   ; clear all but button 3 (mask button3)
(23)          BRNE    main           ; jump if not pressed
(24)          ; ...
(25)  thing:  ; Some other code here
(26)          ;  that does something useful
(27)          ; ...
(28)          BRN     main           ; jump back to main loop
(29)  ;-----------------------------------------------------------------------
```

Figure 14-11: A code fragment showing a typical bit-masking operation using a TEST instruction.

14.5 Comparing Two Values

Often times when dealing with assembly language, you'll have a need to compare two numbers. This is because in many instances, program flow is dependent upon the relationship between the numbers in question. A quick perusal of the RAT instruction set does in fact reveal that there is a compare instruction CMP, which is good. However, the problem is that we sometime want to establish one many equality relationships but we only have a C and Z flag to work with. The relative lack of condition flags does present a problem, but there are some simple solutions. It turns out that you can obtain the relationship between two numbers by using the CMP or SUB instructions and examining the condition flags.

The C flag is actually more versatile than the Z flag. Note that all the shift and rotate instructions used the carry flag in one way or another as the RAT Assembly Manual details. Note that ADD, SUB, and CMP operations modify both the C and Z flags based on the result of those operations. The function of the carry after the ADD instruction should be somewhat intuitive as it indicates that there was a carry out of the 8-bit registers resulting from the addition operation. In this case, the carry acts as its name implies and is simply indicates an overflow from the eight register bits into the carry bit due to the magnitude of the associated operands.

The use of the C flag in the SUB-type instructions is much less intuitive. In this case, the carry flag acts as a "borrow" flag. The borrow flag is set anytime the result of the subtraction operation needs to "borrow" a number from the next significant bit up and outside of the given 8-bits in the register. If you think about it, this is not any different from doing subtraction on paper.

A common use of the carry flag is to establish a relationship between two values. When you use the C flag in conjunction with the Z flag, it is straightforward yet somewhat indirect method to know the relation between two numbers. As you probably know, there can be three relationships between two numbers: less than, equal to, or greater than the other (not including the combinations of these). We can use the condition flags (C and Z) in conjunction with the CMP command to establish this relationship. Table 14.4 shows these relationships; they are somewhat intuitive but we list them here in order to remind you of their existence and availability for use in your assembly programs. Keep in mind the "-" represents a "don't care".

Operation: `CMP rX, rY`		
C	**Z**	**Comment**
0	0	rX > rY
1	0	rX < rY
-	1	rX = rY

Table 14.4: Using the condition flags to establish equality relationships between two values.

14.6 Look-Up Table (LUT) Implementations on the RAT MCU

Generally speaking, anytime you can use a LUT in your hardware or software, you do so. There are many advantages to using a LUT, particularly in firmware applications such as display multiplexing. The RAT instruction set architecture (ISA) was designed specifically to support LUT implementations by using the RAT's scratch RAM and the register-indirect-type load and store instructions (LD and ST).

Using a LUT on the RAT MCU requires three steps. First, you must generate the data that goes into the LUT. Second, you must put the data into the LUT. Third, you must understand how to access the LUT's data using the RAT's instruction set. The following three steps describe the set-up of a LUT for 7-segment display multiplexing.

Step 1) Generate the LUT data: The data you need to put in the LUT is the segment values you need to drive digits 0-9 on a seven-segment display.

Step 2) Put the data into the LUT: The LUT will live on the RAT's scratch RAM, which means you must work with the assembler in order to get the data into the LUT. This step requires that you use the RAT MCU's .DB assembler directive in order to define and initialize the appropriate data. Figure 14-12 shows an example of a program fragment that specifies ten bytes of data that you can use to have the assembler set up the LUT. Recall that each byte of data specified by the .DB option forces the assembler to generate two instructions of start-up code: one MOV instruction to move data into a register, and one ST instruction that stores the data into scratch RAM before the main part of your program begins execution. The assembler automatically inserts the start-up code into program memory when you use the .DB assembler directive.

Figure 14-12 shows a fragment of a RAT assembly program that highlights the approach to defining data for the RAT's scratch RAM. The code in Figure 14-12 uses the .DB directive to declare bytes of data. This data is contiguous starting at address 0x00 in scratch RAM as directed by the numeric value associated with the .ORG directive. Each byte listing in the .DB directive has its own address in scratch RAM. Figure 14-12 shows ten bytes of data being declared; the value of 0x7A will be stored at address 0x00 in scratch RAM, 0x7B will be stored at address 0x01 in scratch RAM, etc.; the value 0x8E will be stored at address 0x09 in scratch RAM. Proper use of the .DB directive and understanding how the assembler uses that data to initialize the scratch RAM values is imperative to writing good RAT assembly language programs.

```
;-------------------------------------------------------------------------
;-- Data Definitions
;-------------------------------------------------------------------------
.DSEG               ; we're in the data segment
.ORG 0x00           ; set the data segment counter to 0x00

;-------------------------------------------------------------------------
; data used to drive segments for decimal numbers
;-------------------------------------------------------------------------
seg_data:    .DB 0x7A, 0x7B, 0x7C, 0x7D, 0x7E  ; for (0-4) not correct data!
             .DB 0x8A, 0x8B, 0x8C, 0x8D, 0x8E  ; for (5-9) not correct data!
```

Figure 14-12: Format for defining LUT.

Step 3) Access the LUT data: After you've successfully completed the previous two steps, you'll need to access the LUT data. Recall that the ST instruction placed the data in scratch RAM as part of the start-up code. Now you must use the RAT's LD instruction to access that data. In particular, you must use the register indirect form of the LD instruction to easily (simple and fast) access the LUT data.

Figure 14-13 shows how you access the data in the LUT under program control. The code first loads a value into a register (neither the value or the register are significant). The code then copies the value associated with the number in register r1 scratch RAM into register r2. The LD instruction uses parenthesis around the r1 argument to indicate indirection. The notion of indirection essentially means that we treat the value in parenthesis as an address; we use this address to access the given value from scratch RAM.

The beauty of this approach for accessing data maybe can only be appreciated by those people who have written extensive code that compares a given value to each decimal number until the appropriate number is found; only then can the appropriate segment data be assigned. The great part about the code in Figure 14-13 is that you never have to write know what number you're dealing with in order to load the segment data associated with that number from scratch RAM. If you're not overly impressed with this approach, you may not be fully understanding it.

```
;-------------------------------------------------------------------------
;-- Code Fragment
;--
;-- Place the data associated with '5' into register r2
;-------------------------------------------------------------------------
        MOV   r1,0x05      ; value in r1 is essentially in BCD format
        LD    r2,(r1)      ; segment data for '5' now in r2
;-------------------------------------------------------------------------
```

Figure 14-13: Code fragment showing how to access data in the RAT MCU's LUT.

14.7 Programming Efficiency of Issues

Out there in computerland, there are always many different approaches to performing the same task. This is also true for assembly language programming. Though there are many different ways to do the same thing and obtain the same result, there are generally underlying differences in the code. This section describes a few of the more obvious issues, which we group into the notion of "programming efficiency".

The term "programming efficiency" certainly sounds good. Suppose you tell your boss that the code you wrote is every efficient. If your boss is actually not just sitting there taking up space, she will ask you, "What makes your code efficient?". The issue here is that there are different forms of efficiency. The two forms we discuss in this section are run-time efficiency and program memory space efficiency. I always think of the example of

knowing an algorithm that I can use to save the world. Sounds good, right? What if the algorithm requires too much program memory space to actually implement? What if you could implement the algorithm in a given amount of code space, but it takes 5000 years to run the code? In these cases, your algorithm is useless[21]. While this is an extreme example, it does nicely describe issues that good programmers make every day (more like every minute).

14.7.1 Iterative Construct Overhead Issues

The underlying problem with iterative constructs is that they have associated "loop overhead". This means that loop constructs contain instructions that don't do anything useful other than maintain the iterative integrity of the construct. This creates a well-known trade-off in coding: fast code vs. less code. The idea behind fast code is that is runs "fast". The idea behind less code is that the code itself takes up more space in the program memory. While we all want our programs to run super-fast, we can't always do that if we're writing code for an environment that has limited program memory.

Figure 14-14 and Figure 14-15 show two subroutines that perform the exact same task: they calculate the parity of a given register and leave that parity information in the C flag. The code in Figure 14-14 does not use an iterative construct while the code in Figure 14-15 does use an iterative construct.

Your first look at these subroutines shows that there are fewer instructions for the code that uses an iterative construct (10 instructions vs. 21 instructions). This means that the code for the iterative construct requires less space in program memory. The execution of these programs tells another store. While the non-iterative subroutine requires 21 instructions to complete, the iterative subroutine requires 39 instructions. The moral of the store is that the non-iterative version of the subroutines requires about twice as much program memory space, but runs twice as fast as the iterative version. This trade-off is something you always need to think about while programming in assembly language. The most easily applied issue associated with this is that you should never use an iterative construct that you know will iterate less than three times[22].

[21] This is a classic technique that many people in academia use to enhance their images. In academia, people seem to get a lot of credit for "good ideas". The truth is that good ideas are meaningless if they are not actually implemented. Needless to say, in academia, there are many more good ideas than there are ideas that people actually took the time to properly implement. Just sayin'.

[22] If you need to do a lot of work in your iterative construct, iteration counts of two are acceptable as it does save program memory space.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Get_par – finds the parity of the data sent in r20. The
(02)  ;   r0 register is set on return for odd parity or cleared for even
(03)  ;   parity.
(04)  ;-
(05)  ;- Affected Reg & flags: C flag
(06)  ;-------------------------------------------------------------------------
(07)  Get_par:
(08)  init:       PUSH    r0
(09)              MOV     r0, 0x00    ; clear register r0
(10)
(11)              LSR     r20         ; shift LSB into C flag
(12)              ADDC    r0,0x00     ; accumulate value of C flag
(13)              LSR     r20         ; etc. (do it 7 more times)
(14)              ADDC    r0,0x00     ;
(15)              LSR     r20         ;
(16)              ADDC    r0,0x00     ;
(17)              LSR     r20         ;
(18)              ADDC    r0,0x00     ;
(19)              LSR     r20         ;
(20)              ADDC    r0,0x00     ;
(21)              LSR     r20         ;
(22)              ADDC    r0,0x00     ;
(23)              LSR     r20         ;
(24)              ADDC    r0,0x00     ;
(25)              LSR     r20         ;
(26)              ADDC    r0,0x00     ;
(27)              LSR     r20         ;
(28)              ADDC    r0,0x00     ;
(29)
(30)              LSR     r0          ; shift LSB into C flag
(31)
(32)  restore:    POP     r0
                  RET                 ; bring it on home
      ;-------------------------------------------------------------------------
```

Figure 14-14: Subroutine that determines parity of register value without using iterative construct.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- subroutine: Get_par – finds the parity of the data sent in r20. The
(02)  ;   r0 register is set on return for odd parity or cleared for even
(03)  ;   parity.
(04)  ;-
(05)  ;- Affected Reg & flags: C flag
(06)  ;-------------------------------------------------------------------------
(07)  Get_par:
(08)  init:       PUSH    r1          ; save r1 value
(09)              PUSH    r0          ; save r0 value
(10)              MOV     r1,0x08     ; iterative count
(11)
(12)  loop:       LSR     r20         ; shift LSB into C flag
(13)              ADDC    r0,0x00     ; accumulate value of C flag
(14)              SUB     r1,0x01     ; decrement loop count
(15)              BRNE    loop        ; branch if not done with loop
(16)
(17)              LSR     r0          ; shift LSB into C flag
(18)
(19)  restore:    POP     r0          ; restore used registers
(20)              POP     r1
(21)              RET                 ; bring it on home
(22)  ;-------------------------------------------------------------------------
```

Figure 14-15: Subroutine that determines parity of register value that uses an iterative construct.

14.7.2 Subroutine Overhead Issues

The underlying problem with subroutine calls is that they also have overhead associated with them. The notion of "overhead" in this context is having the MCU execute an instruction that does not actually do anything

useful for the given task. While we all know that we can use subroutines to keep our code well organized and efficient in terms of program memory, we can also abuse them. This section provides yet another extreme example where we abuse subroutine calls by including them when we really should not have.

Figure 14-16 and Figure 14-17 show two code fragments that perform the exact same task. The code in Figure 14-16 adds a number to a register four times. The code in Figure 14-17 performs the same task, but does so by using a subroutine call. Yes, this is an overly simplified example, but it proves the point.

The code in Figure 14-16 performs the given task in using four instructions. The code in Figure 14-17 performs the same task, but requires a total of 12 instructions, thus requiring three times as much time to perform the same task. The difference in running times of these code fragments has to do with the subroutine call/return overhead. Specifically, each add operation has an associated CALL and RET instruction. These are the instructions that don't do anything except perform administrative issues for the subroutine. Additionally, the code in Figure 14-16 requires less program memory space; it requires four instructions compared to the six instructions of Figure 14-17. The moral of the story is that you should strive to prevent the structure of your program from adding extra running time to your programs.

```
(00)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)  ;
(02)              ADD     r1,0x04             ; add some value
(03)              ADD     r1,0x04             ; etc.
(04)              ADD     r1,0x04             ;
(05)              ADD     r1,0x04             ;
(06)                                          ;
(07)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 14-16: A piece of code without a subroutine.

```
(00)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(01)  ;
(02)              CALL   Add_four       ; do something
(03)              CALL   Add_four
(04)              CALL   Add_four
(05)              CALL   Add_four
(06)                                    ;
(07)  ;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(08)
(09)
(10)  ;-------------------------------------------------------------------------
(11)  ;- subroutine: Add_four – Near meaningless subroutine, but serving as
(12)  ;              an excellent example.
(13)  ;-
(14)  ;- Affected Regs & flags: r1, C flag
(15)  ;-------------------------------------------------------------------------
(16)  Add_four:
(17)               ADD    r1, 0x04     ; save r1 value
(18)               RET                 ; bring it on home
(19)  ;-------------------------------------------------------------------------
```

Figure 14-17: Equivalent code using a subroutine.

14.7.3 Program Space vs. Bullet-Proof Code Issues

As you have probably figured out by now, there are always many approaches to performing the same task when programming computers. You the programmer will always face many subtle but important design decisions when writing your code. This section examines another subtle issue, yet clever opportunity for you to write "appropriate" code.

The code in Figure 14-18 shows the Mult_num subroutine from a previous section. This subroutine multiples the values in two registers and stores the result in another register. The code in Figure 14-18 and Figure 14-19

both perform the same task, but do so using a different amount of code in the subroutines. There is much of the same code in these two subroutines, but there is more code in Figure 14-18.

So I know there is a science and associated mathematics behind this idea, but I don't know what they are. The best I can tell you is that ***in most cases***, the code in Figure 14-19 will usually run faster because there is simply less code. So is "in most cases" good enough? The code in Figure 14-18 will run faster if one of the operands is zero. The code in Figure 14-18 checks both operands before entering into the iterative loop. If one of the operands is zero, the subroutine will return before executing the iterative portion of the subroutine. In these cases, the longer code in Figure 14-18 will run faster. In this case, the extra code effectively shortened the running time of the subroutine. But in the case where both operands are non-zero, the iterative portion of the subroutine executes the iterative portion of the code. In this case, the code at the beginning of the subroutine that checks for zero in both of the operands effectively does nothing.

So which one should you use? It depends. If all the number possibilities for the operand are of equal probability, then a zero will only appear $1/16^{th}$ of the time. In this case, 15/16 of the time, the MCU will execute the iterative portion of the code. In this case, it would probably be better to use the code in Figure 14-19. But if you somehow had inside information on the two operands sent to the subroutine, you may know that a goodly percentage of the time one of the operands would be zero. In this case, you should use the code in Figure 14-18. Moreover, if you know one of the operands would be zero and know nothing about the other, you could save running time by only testing the one operand you knew something about. Or, if you had to check both of the operands, you would check the one you knew something about first. Saving an instruction here and there adds up, particularly when you're running MCU with mega-Hertz clock cycles and your MCU has more important things it could be doing.

```
;-----------------------------------------------------------------------
;- subroutine: Mult_num  - places the effective multiplication
;-  of the number in register r8 with the number in register r9. It
;-  is expected that these number are limited to 4-bits or the
;-  result in register r10 will not be valid.
;-
;- Tweaked regs & flags: r9, r10, C, Z
;-----------------------------------------------------------------------
Mult_num:    MOV     r10,0x00   ; clear register r10
             TEST    r8,0x00    ; set flags
             BRNE    ret1       ; jump to check other operand
             RET                ; return 0 if operand is 0
                                ;
ret1:        TEST    r9,0x00    ; set flags
             BRNE    loop       ; jump to do calculation
             RET                ; return 0 if operand is 0
                                ;
loop:        ADD     r10,r8     ; add multiplicand
             SUB     r9,0x01    ; decrement other operand
             BRNE    loop       ; jump if r9 is not zero (add again)
             RET                ; r9 is zero; result is in r10.
                                ; return control to the main program
;-----------------------------------------------------------------------
```

Figure 14-18: A running-time efficient version of the Mult_num subroutine.

```
;-------------------------------------------------------------------
;- subroutine: Mult_num  - places the effective multiplication
;-  of the number in register r8 with the number in register r9. It
;-  is expected that these number are limited to 4-bits or the
;-  result in register r10 will not be valid.
;-
;- Tweaked regs & flags: r9, r10, C, Z
;-------------------------------------------------------------------
Mult_num:    MOV     r10,0x00   ; clear register r10
                                ;
loop:        ADD     r10,r8     ; add multiplicand
             SUB     r9,0x01    ; decrement other operand
             BRNE    loop       ; jump if r9 is not zero (add again)
             RET                ; r9 is zero; result is in r10.
                                ; return control to the main program
;-------------------------------------------------------------------
```

Figure 14-19: A program memory space efficient version of the Mult_num subroutine.

14.8 Seven-Segment Display Multiplexing

Picture this scenario… Someone in the business department of your company got wind of that fact that the RAT MCU in one of your company's products was being under-utilized. Over an ensuing conversation at the company watering hole, the businessperson convinced the engineering manager to remove the discrete BCD to seven-segment decoder and display multiplexer from the PC board in order to lower the cost of the BOM (bill of materials). The new approach is that the project could use the RAT microcontroller to implement the same functionality. Such a lowering of costs has a way of exciting most business people but it's always the engineers who adjust for these "minor" circuit changes. Because you're the only engineer who knows anything about the RAT, it's your job to implement a firmware display multiplexor on the development board.

There are two main reasons hardware uses seven-segment display multiplexing. First, it saves inputs. Imagine a four-digit 7-segment display that included decimal points. If the displays did not use multiplexing, they would require 32 separate pins (outputs) to properly drive the device. This is many outputs and outputs embedded systems programming considers outputs relatively expensive in embedded designs. A four-digit multiplexed display would require on 12 outputs to drive it. This is because all of the segments are driven simultaneously (a-g and the decimal point) meaning that when you turn on one segment, all the segments are potentially activated. You control which segment actually turns on by actuating the correct gate device associated with each display[23]. The second reason is that you can consider a multiplexed display as requiring less power. This is because only one display is on at a given time[24].

Display multiplexing refers to the general practice of only using one seven-segment decoder to drive multiple BCD inputs to several seven-segment displays. The approach is to actuate each individual seven-segment display sequentially in a circular manner thus ensuring each display activates for the same amount of time. This multiplexing action takes advantage of the human visual system's characteristic of *retinal persistence* in order to make the display appear as if the individual digit displays activate simultaneously while in fact only one digit of the display actuates at a time.

You can implement display multiplexing in either hardware or firmware; this section describes a firmware implementation. Imagine a development board that contains four seven-segment displays; each display contains eight segments (including a decimal point). There is one signal per each segment for all of the four displays on the board; the activation of a single segment on a single digit display involves actuating that segment and actuating the display enable (anode) for that digit. Each individual segment of the seven-segment displays on the

[23] Seven-segment displays come in either "common anode" or "common cathode" configuration, which you can effectivel consider an on/off switch for a given display. Turning on a single segment on a single display requires that you both drive the segment and turn on the proper display.
[24] This is not a super strong reason as was the first, but people often consider it significant.

development boards connect to each other, so writing to one individual segment of a display is actually writing to that segment on all four seven-segment displays

To make the displays appear as if they are constantly on without the appearance of flicker, you need to leave each display actuated for a given amount of time using a firmware delay function such as the one in Figure 14-20. The idea is to do no further processing for a set period of time after a display has been actuated before going on to actuate the next display. The firmware delay in Figure 14-20, however, is not an efficient use of the microcontroller's resources since the program execution is in a tight loop that effectively does no processing. It is, however, a viable firmware-based approach to providing a time delay.

```
;-----------------------------------------------------------------------
;- Subroutine: Delay
;- Description: creates a delay using an iterative construct
;-
;- Tweaked regs & flags: r7, C, Z
;-----------------------------------------------------------------------
Delay:
        MOV     r7,0xFF           ; preset loop count

loop:   SUB     r7,0x01           ; decrement
        BRNE    loop              ; loop if s7 != 0
        RET
;-----------------------------------------------------------------------
```

Figure 14-20: Assembly code for firmware delay.

Figure 14-21 shows a useful display-multiplexing algorithm for a two-digit display. As you can imagine, you could write one continuous section of code to implement this algorithm, but writing code in this manner makes that section of source code specific to driving two seven segment displays. A better approach is to write your code using a modular approach so that you could quickly modify your code to drive three or more seven-segment displays. The diagrams in Figure 14-22 outline a modular approach; these algorithms describe the identical functionality as the algorithm in Figure 14-21 except we separate the original functionality into two modules.

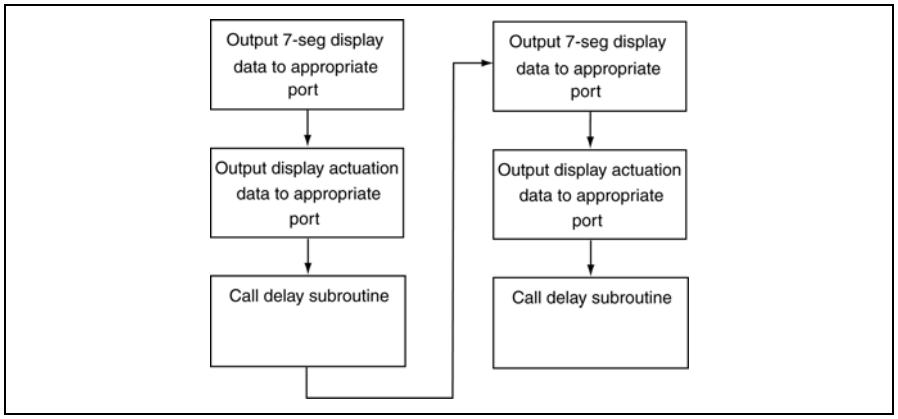
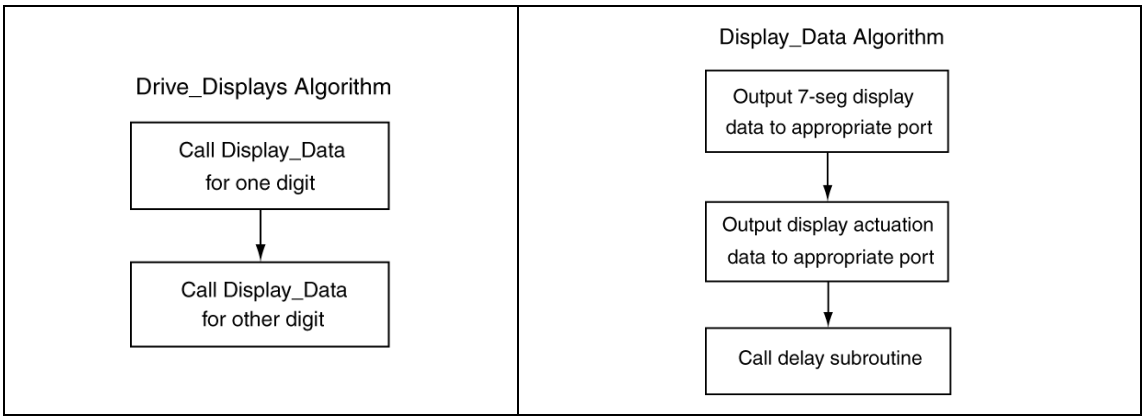


Figure 14-21: Process flow for firmware multiplexing algorithm.

The key to making this modular approach work is to take a generic approach to writing the algorithms in this section. The code that implements the algorithm in Figure 14-22(b) should expect to find data to drive the segments and data to actuate a single display in two different registers. The algorithm in Figure 14-22 (a) should place the proper data in those registers before it calls the algorithm in Figure 14-22(b).



(a) (b)

Figure 14-22: Flowcharts for algorithms that modularize the algorithm shown in Figure 14-21.

14.9 Chapter Summary

- There are three main methods to "pass" value to and from subroutines: 1) registers, 2) C & Z flags, and, 3) scratch RAM memory. No matter how you do it, you should document it in the subroutine header.
- Iterative constructs have underlying issues that programmers need to be aware of. They include types of constructs (while vs. do-while) and off-by-one issues.
- There are four things you can do with a bit, 1) set it, 2) clear it, 3) toggle it, and 4) hold it. Assembly languages set bits by ORing them with '1, clear bits by ANDing them with '0', and toggling bits by EXORing them with '1'. Assembly languages hold bits in many different ways.
- The RAT MCU uses condition flags to determine the equality of two number. Programmers typically use the CMP instruction for such comparisons. If the Z flag is set after a CMP instruction, the two values are equal. If the C flag is set after a CMP instruction, the second operand is greater than the first operand. If the C flag is cleared after a CMP instruction, the first operand is greater than the second operand.
- There are two main efficiency issues in assembly language programming: run-type efficiencies and program memory space efficiencies. The programmer needs to be aware of this trade-off and program their computer appropriately.
- Seven-segment display multiplexing is a method to have seven-segment displays show many numbers but only use a minimal amount of input pins. Input and outputs are expensive on computer devices; the main purpose of a multiplexed display is to reduce the number of I/O required to drive the device. In a multiplexed display, only one digit actuates at any given time. The displays take advantage of the human visual system's retinal persistence to make it appear that all the display devices are on at the same time. It does not have leaving a single digit display on for a given amount of time, then switching to another digit to repeat the process. When this is done fast enough, the human visual system sees more than one number at a given time.

14.10 Chapter Exercises

1) Describe the three ways you can pass data to and from subroutines in the RAT MCU.

2) Describe the main difference between a do-while and while iterative construct.

3) Where do you typically find most errors in assembly language programs?

4) Describe the four things you can do with a single bit.

5) There are standard approaches to setting, clearing, and toggling bits in assembly languages; provide examples of these approaches.

6) Why do MCUs have a need for bit-masking?

7) Describe how the RAT MCU uses flag values to determine equality (or lack thereof).

8) Describe how LUTs can help programmers create efficient code.

9) Describe the difference between accessing a LUT located at address 0x10 and a LUT located at 0x20. For this problem, assume each LUT has ten locations.

10) Describe the two types of programming efficiencies in the RAT MCU assembly language.

11) Describe why assembly code that has more instructions can have a shorter running time than code that has fewer instructions.

12) Describe why are seven-segment display typically multiplexed?

13) Describe the basic operation of a multiplexed display.

15 RAT Programming Problems

15.1 Introduction

The only way to learn about assembly language programming is to actually do some assembly language programming. The previous chapters spoke about the basic mechanics of assembly language programs, but only provided a few basic examples. This chapter presents nothing new, but presents all of the older assembly language programming ideas in the context of example problems. The problems start out easy and become more challenging as the chapter progresses. The idea here is that if you understand all the example programs in this chapter, then you'll know about all the tricks associated with assembly language program. Keep in mind that one of my theories of assembly language programming that if you see and understand a trick once, you can put that trick in your bag of tricks and whip it out whenever you need it. It's that easy.

Main Chapter Topics

- **NO NEW TOPICS:** This chapter presents all previously presented stuff in the context of actual example programs.

Why This Chapter is Important

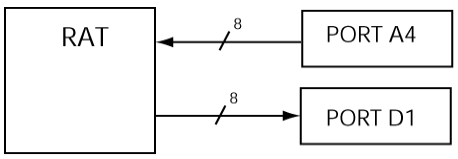
This chapter is important because it shows how to solve a wide set of problems by writing RAT assembly language programs.

15.2 Introductory RAT Programming Problems

Here are a few introductory RAT programming problems. Each solution contains pertinent highlights as well as the well-commented source code.

Example 15-1:

Write a RAT assembly language program that reads data from the listed input port. Take the 2's complement of the input value and output it to the listed output port. Perform this operation repeatedly.



Solution: There are a few assumptions we're making regarding this problem and subsequent solution. Since this is the first RAT MCU programming problem we've done, we'll explicitly note some of finer details of this problem.

- The diagram shows that the RAT MCU interfaces with two I/O devices: one input device and one output device. We know this because of the direction of the arrows; the device with the arrows exiting the device and entering the RAT MCU is an input device and the device with the arrows exiting the RAT MCU and entering the external device is an output device. In this context, input devices provide information to the RAT (such as status of the device) while output devices provide a path for data to flow out of the RAT MCU (such as control signals to control devices).

- The diagram lists the port addresses associated with the two external devices. Recall that these are necessarily 8-bit values; the diagram is choosing to list them in hex format. These numbers are arbitrary for this problem; in real life (whatever that means), these values are associated with the hardware, thus someone must provide you with these addresses (or with a datasheet that indicates these addresses) in order for you to know how to interface with these devices.

```
(00) ;-----------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-----------------------------------------------------------------
(03) .EQU PORT_A4 = 0xA4   ; input port
(04) .EQU PORT_D1 = 0xD1   ; output port
(05) ;-----------------------------------------------------------------
(06) .CSEG                 ; indicates code segment
(07) .ORG   0x10           ; sets the code segment counter to 0x10
(08)
(09) init:                           ; nothing to initialize
(10)
(11) main:     IN      r1,PORT_A4    ; get data from input port
(12)           EXOR    r1,0xFF       ; compliment data (1'1 compliment)
(13)           ADD     r1,0x01       ;  then add 1 for 2's compliment
(14)           OUT     r1,PORT_D1    ; output data to specified port
(15)           BRN     main          ; stir and repeat
(16) ;-----------------------------------------------------------------
```

Figure 15-1: Solution for Example 15-1.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution has several of the main parts of an assembly language program, but other parts are missing in one form or another.

    - Though this is a complete program, we've opted to eject the file banner, which generally includes a description of the program, the author's name, and credit cards numbers of people the author does not like.
    - We have arranged the assembler directives towards the front of the program, which is another common good practice.
    - The text appears relatively neat. Note that we line up most of the items rather nicely and we use that same alignment consistently throughout the program. Most instructions have comments. Some sections have either banners or use labels to indicate a special section of code; the "init" and "main" labels are labels that should appear in every assembly language program.
    - The program is rather simple, so the init portion of the program is empty; the "init" label is there for completeness, which is yet another great practice. The idea here is that if the label was not there, someone reading your program would wonder if you forgot to include initialization code. Note that no instructions actually use the "init" label; this common commenting practice is one that you should adopt for your programs.
    - The program does indeed contain a main part, which follows the "main" label.

- Before we write any code in the solution, we first use the "CSEG" and "ORG". The CSEG directive informs the assembler that we'll be writing instructions, or "code", (as opposed to defining data). Any time we use the CSEG directive, we also should use the ORG directive. If you do not use the ORG directive, the assembler could place your code anywhere in the code segment. Using the ORG directive tells the assembler where to place your code in instruction memory so there is no need to wonder where in the code segment the assembler places your code.

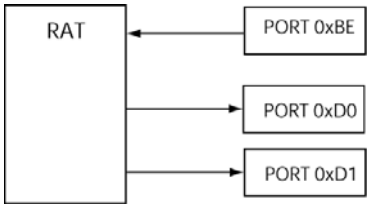
- The solution opts to use .EQU assembler directives in order to represent the port addresses. This is arbitrary but we consider this good practice in assembly language programming. The thought here is that these port addresses are constants in that they generally do not change through the course of the program; in the case that they do change, supporting the change is a matter of changing the assembler directive rather than all the individual uses of the port addresses throughout your program.

- The program uses an EXOR function to perform the requested compliments of the input data. This is typical practice in assembly language programming in that there is no compliment instruction in the RAT instruction set so you have to use what you have, which is the EXOR function.

- The use of register "r1" is arbitrary; we could have used any of the RAT MCU's 32 registers. This program advertises the idea that the RAT MCU's 32 registers are truly "general purpose".

Example 15-2:

Write a RAT assembly language program that inputs a value from input port 0xBE. If the input value is less than 0xC4, complement the value and output the result to output port 0xD0; otherwise write the value of 0x55 to output port 0xD1. The program performs this operation repeatedly.



Solution:

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU PORT_BE = 0xBE   ; input port
(04)  .EQU PORT_D0 = 0xD0   ; output port
(05)  .EQU PORT_D1 = 0xD1   ; output port
(06)  .EQU CMP_VAL = 0xC4   ; important comparison value
(07)  .EQU OUT_VAL = 0x55   ; important output value
(08)  ;-------------------------------------------------------------------------
(09)  .CSEG                         ; indicates code segment
(10)  .ORG   0x01                   ; sets the code segment counter to 0x01
(11)
(12)  init:       MOV     r31,OUT_VAL    ; pre-store output value
(13)
(14)  main:       IN      r0,PORT_BE     ; get data from input port
(15)              CMP     r0,CMP_VAL     ; compare register contents r0-0xC4
(16)              BRCS    less_than      ; jump if C=1 (sub generated carry)
(17)              OUT     r31,PORT_D1    ; output specified value
(18)              BRN     main
(19)
(20)  less_than:  EXOR    r0,0xFF        ; compliment input value
(21)              OUT     r0,PORT_D0     ; output result to out port
(22)              BRN     main           ; stir and repeat
(23)  ;-------------------------------------------------------------------------
```

Figure 15-2: Solution for Example 15-2.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- We model the solution as an if/else construct; the code either does one thing or another based on the value input from the given input port.

- The initialization portion of this program consists of placing a value into a register. We do this because the OUT instruction only outputs data from a register; by placing that value in a register in the initialization section, we won't have to use a MOV instruction in the main code. This saves an instruction in the main code so the program runs slightly faster, but we also have one less general purpose register to use since we're essentially "storing" a value in r31 for the entirety of the program. This of course makes no difference for short meaningless programs such as this one.

- The solution opts to represent all input and output ports using the EQU directive; this is good practice for all "constants" in the assembly language program. Generally speaking, some external entity pre-defined the port addresses and most likely won't change in your lifetime (or the lifetime of the hardware you're programming). Recall that the assembler directives are not executable code.

- The CMP instruction subtracts the constant value from the value that was input into r0 from the previous IN instruction. If the input value is less than the constant value, a carry is generated (recall that the CMP instruction performs a subtraction operation). Line 16 has the branch instruction; the BRCS only branches when the carry flag is set; otherwise, the program does not take the branch and proceeds to execute the instruction following the BRCS.

- The "less_than" label uses self-commenting to indicate what happens when the input value is "less than" the constant value. In this case, the input value is complimented using an EXOR instruction (line 20) and output to the appropriate port (line 21)

Example 15-3:

Write a RAT assembly language subroutine that multiplies the number in register r6 by 7 (seven) and stores the result in r9. Assume that the program calling this subroutine is responsible for placing the value into register r6. If at any time an addition operation generates a carry, the result overflows the 8-bit RAT register. If the operation exceeds the 8-bit range, indicate the value in r9 is not valid by moving 0xFF into r10; if the operation does not exceed the range, indicate the answer in r9 is valid by moving 0x80 in register r10.

Solution:

```
(00)  ;-----------------------------------------------------------------
(01)  ;- subroutine: Mult_7x
(02)  ;-
(03)  ;- Multiplies number in r6 by 7. If any addition operation generates
(04)  ;- a carry, the result in r9 is not valid and is indicated by placing
(05)  ;- 0xFF in r10; otherwise the result r9 is returned after placing 0x80
(06)  ;- r10.
(07)  ;-
(08)  ;- Affected regs & flags: r9, r1, r10, C, Z
(09)  ;-----------------------------------------------------------------
(10)
(11)  Mult_7x:
(12)  init:       MOV     r9, 0x00       ; clear accumulator
(13)              MOV     r1, 0x07       ; set iteration count
(14)                                     ;
(15)  loop1:      ADD     r9, r6         ; add value to accumulator
(16)              BRCS    not_valid      ; branch if carry generated
(17)              SUB     r1,0x01        ; decrement iterative count
(18)              BRNE    loop1          ; check iterative count
(19)              MOV     r10,0x80       ; set valid flag
(20)              RET                    ; valid return
(21)
(22)  not_valid:  MOV     r10,0xFF       ; set not-valid flag
(23)              RET                    ; not-valid return
(24)  ;-----------------------------------------------------------------
```

Figure 15-3: Solution for Example 15-3.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- This problem asks you to write a subroutine that does something special. That being the case, the code we write does not have to constitute a complete assembly language program. On the other hand, the subroutine is sort of like a miniature program in several ways. First, the subroutine has a complete header providing the user of the subroutine with useful information. Second, it also has initialization code at the beginning of the subroutine, though we typically do not use an "init" label as we do with complete programs. There will be more comment on this in a later comment.

- The subroutine header has three forms of information. First, it contains the name of the subroutine. Next it contains a description of the subroutine; this description is as short but as complete as possible. Recall that only academic administrators score points for being overly verbose, which generally is an attempt to disguise their incompetence. Lastly, the subroutine banner contains a list of the registers that the subroutine changes. This is important as some other part of the program may be using these general purpose registers. Notifying the programmer, which registers the subroutine changes, is great and required programming practice; it will save your ass someday. More comments on this later also.

- The subroutine name is nothing more than an official RAT MCU label. Recall that there is a number associated with each label in a program; the assembler associates a number with the label 'Mult_7x"; this label is a 10-bit value that represents the first instruction in the subroutine, which in this case is a MOV instruction.

- The subroutine also has some initialization information; this starts at the instruction with the "init" label. All RAT MCU assembly programs and individual subroutines have labels indicating some type of initialization is taking place.

- The RAT MCU does not have a multiply instruction. This being the case, we must resort to implementing multiplication by repeated additions. The standard way to do this is to implement the main portion of the subroutine code as an iterative loop. In this case, we know how many times we need to iterate (seven). The body of the loop comprises of performing the addition, checking the results of the addition for a carry, and decrementing the iteration count.

- Because we are using an iterative loop and keeping track of the iteration count, we absolutely need to initialize both the register we use as an iteration counter and the register we use to "accumulate" the result. Once again, most every subroutine will require some form of initialization; this subroutine requires these two items. The notion of an "accumulator" is common in all forms of programming including assembly language; the notion here is that a register is first cleared (the value in that register is set to 0x00) and then we continually add values to that register in another part of the iterative loop.

- This subroutine has the notion of a "return value", which is a sematic artifact from higher-level programming languages. The only way an assembly program has the ability to "return" something (by something, we mean data of some type), is to leave the value in a register. The person writing the program that calls the subroutine knows in which register to find the value. You sometimes see this mechanism referred to as "passing value", where the RAT MCU can only pass values to and from subroutines by leaving those values in registers. This of course emphasizes the notion that you must completely document your subroutine so programmers know everything they can expect from your subroutine.

- This subroutine uses the notion of a "flag", which indicates the status of the multiplication. In the case of this subroutine, the calculation may overflow the 8-bit RAT register, so the subroutine indicates this by writing the specified values to the specified register (which in this case is r10). Assembly language uses flags such as this one quite often. A flag can be anything that transfers/reports status, anywhere from a single bit in a register to the entire 8-bit register value.

- This subroutine has two "exit points", which are RET instructions. Subroutines have at most one entry point but can have as many exit points as necessary to implement the given algorithm in the subroutine.

- After every addition operation (line 16), the subroutine checks to see if the instruction generated an overflow. The subroutine does this by using a conditional branch instruction BRCS. The notion here is that if an addition instruction generates a carry, the result in the accumulator register (r9) will not be valid. In this case, the subroutine has no reason to continue in the loop, so it set the flag register as invalid and then exits the subroutine.

An important alternate solution for this example:

```
(00) ;-----------------------------------------------------------------------
(01) ;- subroutine: Mult_7x
(02) ;-
(03) ;- Multiplies number in r6 by 7. If any addition operation generates
(04) ;- a carry, the result in r9 is not valid and is indicated by placing
(05) ;- 0xFF in r10; otherwise the result r9 is returned after placing 0x80
(06) ;- r10.
(07) ;-
(08) ;- Affected regs & flags: r10, C, Z
(09) ;-----------------------------------------------------------------------
(10)
(11) Mult_7x:
(12) init:       PUSH r9               ; save registers used by subroutine
(13)             PUSH r1
(14)
(15)             MOV     r9, 0x00      ; clear accumulator
(16)             MOV     r1, 0x07      ; set iteration count
(17)                                   ;
(18) loop1:      ADD     r9, r6        ; add value to accumulator
(19)             BRCS    not_valid     ; branch if carry generated
(20)             SUB     r1,0x01       ; decrement iterative count
(21)             BRNE    loop1         ; check iterative count
(22)             MOV     r10,0x80      ; set valid flag
(23)             BRN     restore       ; branch to reg restoration
(24)
(25) not_valid:  MOV     r10,0xFF      ; set not-valid flag
(26) restore:    POP     r1            ; restore registers used by
(27)             POP     r9            ;  subroutine
(28)             RET                   ; standard return
(29) ;-----------------------------------------------------------------------
```

Figure 15-4: Alternate solution for Example 15-3.

Notes Regarding the Alternate Solution: This is the same solution but it does something that is extremely common and important in assembly language programming.

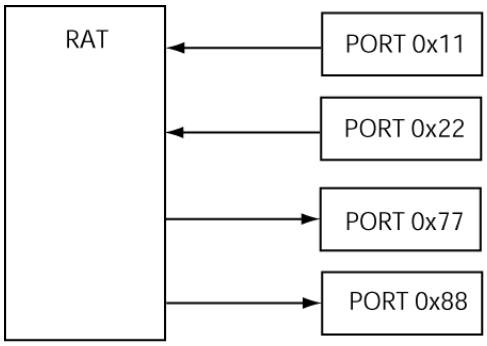
With any CPU, you always have to have good bookkeeping practices regarding your use of the registers. This means when you call a subroutine or process an interrupt, you need to make sure the code in the subroutine or interrupt service routine does not change a register that the main program is currently using. The first solution to this example directly stated in the subroutine header what registers the subroutine changes; this is thus a message from the programmer who wrote the subroutine to the programmer who is calling the subroutine. In this manner, the calling programmer must then be careful not to be using registers that subroutine changes. The alternative solution takes another approach.

Upon entering the subroutine, the first set of code saves the registers that the subroutine alters. The standard approach to doing this is to write push the registers on the stack at the beginning of the subroutine and then pop them off the stack before the subroutine exits. This is a massively common use for the PUSH and POP instructions in is common for any assembly language. Here are the pros and cons of this approach:

> **PROs:** This approach frees the calling programmer from worrying about low-level implementation details in the subroutine. Since the subroutine automatically saves and restores the registers as part of the subroutine, the calling programmer is free not to worry about the subroutine stepping on registers that the calling code is currently using.
>
> **CONS:** The size of the program increases to handle the PUSH and POP instructions. In this example, there are only two registers that required saving so there was not that many more extra instruction; but this would have been a different story if there were ten registers that required saving. The extra instructions both take up code space and create overhead for the subroutine. In this case, the overhead effectively increases the effective running time of the subroutine. This increase in running time becomes more significant the more often the subroutine executes.

Example 15-4:

Write a RAT assembly language program inputs one value from input ports 0x11 and 0x22. If the two input values are equivalent, the program outputs a 0xFF to output port 0x77; otherwise, the program outputs the value 0xAA to output port 0x88. The program performs this operation repeatedly.



Solution:

```
(00)    ;-----------------------------------------------------------------
(01)    ;- Assembler Directives
(02)    ;-----------------------------------------------------------------
(03)    .EQU PORT_11 = 0x11   ; input port
(04)    .EQU PORT_22 = 0x22   ; input port
(05)    .EQU PORT_77 = 0x77   ; output port
(06)    .EQU PORT_88 = 0x88   ; output port
(07)    ;-----------------------------------------------------------------
(08)    .CSEG                         ; indicates code segment
(09)    .ORG   0x01                   ; sets the code segment counter to 0x01
(10)
(11)    init:     MOV     r30,0xFF       ; pre-store value
(12)              MOV     r31,0xAA       ; pre-store value
(13)
(14)    main:     IN      r0,PORT_11     ; get data from input port
(15)              IN      r1,PORT_22     ; get data from input port
(16)              CMP     r0,r1          ; compare register contents
(17)              BRNE    not_eq         ; jump if not equal
(18)              OUT     r30,PORT_77    ; output specified value
(19)              BRN     main
(20)
(21)    not_eq:   OUT     r31,PORT_88    ; output specified value
(22)              BRN     main           ; stir and repeat
(23)    ;-----------------------------------------------------------------
```

Figure 15-5: Solution for Example 15-4.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution opts to represent all input and output ports using the EQU directive; a fine practice indeed.
- The solution starts the code at location 0x010 in the code segment by using the CSEG and ORG directives. The starting point of the code is arbitrary in this example.
- The initialization portion of the code consists of placing values the program will use into registers. This approach saves a few instructions in the main code.

- The body of the code is a classic if/else construct. The program firsts inputs the values from the designated port addresses and then compares them using the CMP instruction (line 16). Based on the result of the compare instruction, if the values are equal, the BRNE instruction fails and drops through to the OUT instruction (line 18); or else, the values are equal and the program takes branch associated with the BRNE instruction, which causes it to branch to line 21.

Example 15-5:

Write a RAT assembly language subroutine that sorts two values. The values are sent to the subroutine in register r30 and r31; in other words, the program that calls this subroutine must place the two values in the register prior to calling the subroutine. The subroutine considers the values in these registers to be unsigned 8-bit numbers. The subroutine places the larger number in register r31 and the smaller number in register r30.

Solution:

```
(00)  ;-------------------------------------------------------------------
(01)  ;- subroutine: Sort_2
(02)  ;-
(03)  ;- Performs a sort on two values sent in r30 & r31. The larger of the
(04)  ;- two numbers is placed in r31; the smaller in r30.
(05)  ;-
(06)  ;- Affected regs & flags: possibly r30 & r31, C, Z
(07)  ;-------------------------------------------------------------------
(08)
(09)  Sort_2:
(10)  init:       PUSH    r29           ; save r29, a working register
(11)
(12)              CMP     r31,r30       ; compare sent values
(13)              BRCS    swap          ; swap if r30 is greater
(14)              BRN     restore       ; r31=r30 or r31>r30; do nothing
(15)
(16)  swap:       MOV     r29,r31       ; temp save of r31
(17)              MOV     r31,r30       ; copy r30 value to r31
(18)              MOV     r30,r29       ; restore original r31 value
(19)
(20)  restore:    POP     r29           ; restore r29
(21)              RET                   ; all done
(22)  ;-------------------------------------------------------------------
```

Figure 15-6: Solution for Example 15-5.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- This is a classic sorting problem. Sorting problems are quite common in computer programming in general, as well as being quite common in assembly language programming. Many algorithms out there require some type of sorting, so subroutines such as this one are quite useful to have around.

- The subroutine uses r29 as a working register, which means the subroutines uses the register but does not care about the value after the subroutine completes. This being the case, the subroutine pushes r29 onto the stack at the beginning of the subroutine and pops it off the stack at the end of the subroutine.

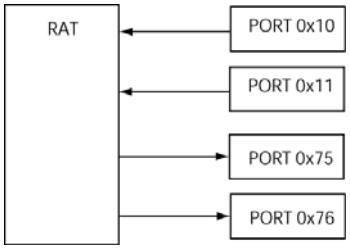
- The heart of this subroutine is the compare of the two passed values in r30 & r31. Line 12 has the comparison instruction; if r30 is greater than r31, the carry flag is set and the subroutine needs to swap the contents of these two registers. Otherwise, the subroutine needs to do nothing except restore r29. Note that this is an if/else construct.

- Note how the solution aligns everything in the solution and the subroutine has a general neat appearance. Also, note that there are blank lines between "sections" of the subroutine. We refer to slightly different pieces of code in this solution; the blank lines make the overall code more readable and understandable.

- The labels in the subroutine use self-commenting names for the labels. The "swap" label associates with the swapping portion of the code while the "restore" label associates with the restoration of the previously saved r29 register.

Example 15-6:

Write a RAT assembly language subroutine that does the following. The subroutine inputs a value from port 0x10; if that value is non-zero, a new value is input from port 0x11, complemented, and output to port 0x75. If the value input from port 0x10 is zero, the value of 0x89 is output to port 0x76. Additionally, if the value input from port 0x10 is non-zero, 0xFF is moved to r27; otherwise r27 is cleared.



Solution:

```
(00) ;-------------------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-------------------------------------------------------------------------
(03) .EQU PORT_10 = 0x10   ; input port
(04) .EQU PORT_11 = 0x11   ; input port
(05) .EQU PORT_75 = 0x75   ; output port
(06) .EQU PORT_76 = 0x76   ; output port
(07) ;-------------------------------------------------------------------------
(08) .CSEG                      ; indicates code segment
(09) .ORG   0x01                ; set code segment counter to 0x01
(10)
(11) init:       MOV     r31,0x89  ; pre-store value
(12)
(13) main:       CALL    my_sub    ; driver program for simulator
(14)             BRN     main
(15) ;-------------------------------------------------------------------------
(16)
(17) ;-----------------------------------------------------------------
(18) ;- subroutine: my_sub
(19) ;-
(20) ;- Inputs value and acts accordingly (not worth writing about).
(21) ;-
(22) ;- Affected regs & flags: r27, C, Z
(23) ;-----------------------------------------------------------------
(24) my_sub:
(25) init:       PUSH    r0             ; save r0
(26)
(27)             IN      r0,PORT_10     ; get data from input port
(28)             CMP     r0,0x00        ; compare register contents r0-0x00
(29)             BREQ    eq_zero        ; jump if input value is 0
(30)
(31)             IN      r0,PORT_11     ; grab new value
(32)             EXOR    r0,0xFF        ; compliment value
(33)             OUT     r0,PORT_75     ; output value
(34)             MOV     r27,0xFF       ; do specified work
(35)             BRN     restore        ; branch to exit
(36)
(37) eq_zero:    OUT     r31,PORT_76    ; output retarded value
(38)             MOV     r27,0x00       ; clear specified register
(39)
(40) restore:    POP     r0             ; restore r0
(41)             RET                    ; end the pain of this subroutine
(42) ;-----------------------------------------------------------------
```

Figure 15-7: Solution for Example 15-6.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- Though this subroutine requested a subroutine, the solution includes both the requested subroutine and what we refer to as a "driver program". We provide the driver program just in case you want to run the solution in the RAT simulator.

- The problem describes an if/else statement; the solution is thus follows suit with an implementation of an if/else construct. There is more than one action for both the "if" and the "else" part of the statement, which you can gather from careful reading of the problem.

- The subroutine uses the stack as a storage device for the register the subroutine changes as part of the subroutine code. In this way, the subroutine stores r0 upon entering the subroutine (line 25) and then restores is as the last instruction before exiting (line 40).

Example 15-7:

Write a RAT assembly language subroutine that does the following. The subroutine inputs a value from port 0x24. If the value is less than 50, the value is added to an accumulator. This subroutine performs this action until the accumulator overflows. The number of additions before the subroutine made before the overflow is placed in r20. Consider all input values to be in unsigned binary form.

Solution:

```
(00) ;-----------------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-----------------------------------------------------------------------
(03) .EQU PORT_24 = 0x24   ; input port
(04) .EQU CMP_VAL = 0x32   ; generic compare value (50 decimal)
(05) ;-----------------------------------------------------------------------
(06) .CSEG                 ; indicates code segment
(07) .ORG   0x01           ; set code segment counter to 0x01
(08)
(09) init:                 ; nothing to do here
(10)
(11) main:       CALL    my_sub    ; driver program for simulator
(12)             BRN     main
(13) ;-----------------------------------------------------------------------
(14)
(15) ;-----------------------------------------------------------------
(16) ;- subroutine: my_sub
(17) ;-
(18) ;- Inputs value and acts accordingly (not worth writing about). The
(19) ;- final value in r1 is the number of time the subroutine can add
(20) ;- input values less than 50 to the accumulator.
(21) ;-
(22) ;- Affected regs & flags: r1 (used as a counter), C, Z
(23) ;-----------------------------------------------------------------
(24) my_sub:
(25) init_1:     PUSH    r0             ; save r0 (accumulator)
(26)             PUSH    r2             ; save r2 (working register)
(27)
(28)             MOV     r0,0x00        ; clear accumulator
(29)             MOV     r20,0x00       ; clear counter register
(30)
(31) in_val:     IN      r2,PORT_24     ; get data from input port
(32)             CMP     r2,CMP_VAL     ; compare register contents
(33)             BRCC    in_val         ; jump if input is > 50
(34)
(35)             ADD     r0,r2          ; accumulate new value
(36)             BRCS    restore        ; quit if add generates carry
(37)             ADD     r20,0x01       ; increment count
(38)             BRN     in_val         ; branch to input new value
(39)
(40) restore:    POP     r2             ; restore saved values
(41)             POP     r0
(42)
(43)             RET                    ; please go away
(44) ;-----------------------------------------------------------------
```

Figure 15-8: Solution for Example 15-6.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- This solution uses two standard constructs: an if/else construct and an iterative construct where the solution does not know the number of iterations in advance. The if/else construct controls whether the value is added or not; it is added when the input value is less than 50 (0x32). The iterative construct

handles the addition portion of the code; the loop continues until an addition operation generates a carry.

- The subroutine has a special "init" label. Since the program uses the "init" label as part of the main code, the subroutine must use a label other than "init". The choice we make here is "init_1", which does in fact convey the important "start of something new" information.

- The solution also records the number of addition operations made in the iterative construct that do not cause the generation of a carry.

- The solution protects the two registers it uses (r0 & r2) by pushing them onto the stack at the beginning of the subroutine and popping them off the stack before the subroutines returns to the calling code.

15.3 C Code-Based RAT Programming Problems

This section contains problems that relate to basic C coding principles and constructs. While this is not an exhaustive list, it does contain some of the more basic and important C constructs and subsequently shows their relation to the underlying RAT assembly language.

Example 15-8:

Write RAT assembly language code that implements the following C programming construct. Assume that r10 holds the "A" value, and r13 holds the "B" value.

```
#define  VAL   48

for (i = 0; i < VAL; i++)  {
   A += B;
}
```

Solution: The code in the example is not a complete program, so the solution is not a complete program either. Both sets of code are examples of code fragments of C code (for the original problem) and RAT assembly code (for the solution).

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives (somewhere in the program)
(02)  ;-------------------------------------------------------------------------
(03)  .EQU VAL     = 0x30   ; random stupid value (for decimal 48)
(04)  ;-------------------------------------------------------------------------
(05)
(06)
(07)  init:       MOV     r31,0x00       ; initialize the iterative count
(08)
(09)  loop:       CMP     r31,VAL        ; compare register to constant
(10)              BREQ    done           ; end loop if they are equal
(11)              ADD     r10,r13        ; accumulate A variable
(12)              ADD     r31,0x01       ; increment by 1
(13)              BRN     loop           ; jump to attempt new iteration
(14)
(15)
      done:                              ; code breaks out of loop
      ;-------------------------------------------------------------------------
```

Figure 15-9: Solution for Example 15-9.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution uses an iterative construct where the program knows the number of loop iterations before it executes the code fragment.
- The problem statement gives the registers where the assembly language programs stores the important values. The "c_cnt" value is the initial value of the iterator. The "VAL" value seems like a constant based on the full capitalization so we the solution uses an assembler directive to represent this value.
- No initialization is necessary for the "A" and "B" variables. The solution assumes the registers already hold the correct values, which is a safe assumption to make for this example.
- The "A" variable acts as an accumulator according to the "A += B" code (recall that "A += B" is a C code shorthand notion meaning "A = A + B"). The problem continues in the loop until the iterator value equals the number of iterations completed by the loop.
- The "done" label is arbitrary; it primarily serves as a comment for the code fragment in this solution.

Example 15-9:

Write RAT assembly language code that implements the following C programming construct. Assume r8 holds the "c_cnt" value, r10 holds the "A" value, and r13 holds the "B" value.

```
#define  VAL   48

for (i = c_cnt; i < VAL; i+=2)  {
   A += B;
}
```

Solution: The code in the example is not a complete program, so the solution is not a complete program either. Both sets of code are examples of code fragments of C code (for the original problem) and RAT assembly code (for the solution). This problem is eerily similar to the previous problem, so it is important you realize the differences, as they are rather special and somewhat tricky.

```
(00) ;-------------------------------------------------------------------------
(01) ;- Assembler Directives (somewhere in the program)
(02) ;-------------------------------------------------------------------------
(03) .EQU VAL     = 0x30   ; random stupid value (for decimal 48)
(04) ;-------------------------------------------------------------------------
(05)
(06) init:       MOV     r31,r8         ; copy the iterative value
(07)
(08) loop:       CMP     r31,VAL        ; compare register to constant
(09)             BRCC    done           ; jump if C=1 (sub generated carry)
(10)             ADD     r10,r13        ; accumulate A variable
(11)             ADD     r31,0x02       ; increment by 2
(12)             BRN     loop           ; jump to attempt new iteration
(13)
(14) done:                              ; code breaks out of loop
(15) ;-------------------------------------------------------------------------
```

Figure 15-10: Solution for Example 15-9.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution uses an iterative construct where the program does not know the number of iterations in advance. This differs from the previous problem in that the fragment of code does not know the "c_cnt" value at compile time (though it necessarily knows the value at run time).

- The problem statement gives the registers where the assembly language programs stores the important values. The "c_cnt" value is the initial value of the iterator. The "VAL" value is a constant based on the full capitalization so we the solution uses an assembler directive to represent this value. The problem states that the iterator variable increments by two each iteration.

- The solution copies the value of "c_cnt" into another register in order to protect the original value of "c_cnt". This is arbitrary; the compiler may or may not actually insert code such as this.

- No initialization is necessary for the "A" and "B" variables. The solution assumes the registers already hold the correct values, which is a safe assumption to make for this example.

- The problem continues in the loop until the increment by two operation (line 11) causes the iterative value in r31 to exceed the value of VAL; the CMP operation (line 08) generates a carry when this occurs. The generated carry causes the code to take the branch (line 09); otherwise, the loop continues.

Example 15-10:

Write RAT assembly language code that implements the following C programming construct. Assume r18 holds "a_val" and r25 holds "sensor_01".

```
#define  C_DUB      192
#define  RESET_VAL   65
#define  INIT_VAL    80
#define  INC_VAL      3


if (a_val == C_DUB) {
  sensor_01 = RESET_VAL;
}
else  {
  sensor_01 = INIT_VAL + INC_VAL;
}
```

Solution:

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU C_DUB     = 0xC0   ; idiot value
(04)  .EQU RESET_VAL = 0x41   ; durf value
(05)  .EQU INIT_VAL  = 0x50   ; retard value
(06)  .EQU INC_VAL   = 0x03   ; derixon value
(07)  ;-------------------------------------------------------------------------
(08)
(09)  init:       CMP     r18,C_DUB      ; do initial comparison
(10)              BRNE    not_eq         ; output value
(11)              MOV     r25,RESET_VAL  ; “if” portion: assign sensor_01
(12)              BRN     done           ; jump over else instructions
(13)
(14)  not_eq:     ADD     r25,INIT_VAL   ; implements the “else” part of C code
(15)              ADD     r25,INC_VAL    ; do even more stupid stuff
(16)
(17)  done:                              ; self-commenting label
(18)  ;-------------------------------------------------------------------------
```

Figure 15-11: Solution for Example 15-10.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution uses .EQU assembler directives for the associated compiler directives (the #defines) in the C code. This is a good choice.
- An equality statement drives the “if” portion of the C code. The solution implements this equality check using a CMP instruction (line 09).
- If the two operands in the CMP instruction are not equal, the code branches to the “else” portion of the code, starting at line 14. If the two operands are equal, the solution executes the instructions associated with the “if” part of the C code, which is line 11 in the solution. The “if” portion of the code then jumps over the “else” portion of the code (lines 14-15) to the code associated with (or following) the “done” label.

Example 15-11:

Write RAT assembly language code that implements the following C programming construct. Consider all variables to be declared as unsigned chars. Assume r10 holds "val", r11 holds "a_val", r12 holds "b_val", and r13 hold "c_val".

```
switch (val)
{
   case 0x01:
      a_val++;

   case 0x08
      b_val++;

   case 0x02:
   case 0x00:
      c_val++;

   default:
      a_val = 0;
}
```

Solution: Once again, the code in the example is not a complete program, so the solution is not a complete program either. Both sets of code are examples of code fragments of C code (for the original problem) and RAT assembly code (for the solution).

```
(00) ;-----------------------------------------------------------------------
(01) ;- Assembler Directives (somewhere in the program)
(02) ;-----------------------------------------------------------------------
(03) .EQU ONE    = 0x01   ; value (for decimal 1)
(04) .EQU EIG    = 0x08   ; value (for decimal 8)
(05) .EQU TWO    = 0x02   ; value (for decimal 2)
(06) .EQU ZER    = 0x00   ; value (for decimal 0)
(07) ;-----------------------------------------------------------------------
(08) init:                            ; nothing to init
(09)
(10) test_1:     CMP     r10,ONE      ; compare register to ONE constant
(11)             BRNE    test_8       ; branch to next test (not equal)
(12)             ADD     r11,0x01     ; increment a_val
(13)             BRN     done         ; stop looking
(14)
(15) test_8:     CMP     r10,EIG      ; compare register to EIG constant
(16)             BRNE    test_2_0     ; branch to next test (not equal)
(17)             ADD     r12,0x01     ; increment a_val
(18)             BRN     done         ; stop looking
(19)
(20) test_2_0:   CMP     r10,TWO      ; compare register to TWO constant
(21)             BREQ    incr_c       ; branch if same value
(22)             CMP     r10,ZER      ; compare register to ZER constant
(23)             BREQ    incr_c       ; branch is values are the same
(24)             BRN     default      ; branch to default case
(25)
(26) incr_c:     ADD     r13,0x01     ; increment a_val
(27)             BRN     done         ; stop looking
(28)
(29) default:    MOV     r11,0x00     ; default case for switch statement
(30)
(31) done:                            ; code exits construct
(32) ;-----------------------------------------------------------------------
```

Figure 15-12: Solution for Example 15-11.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- A quick perusal of the solution highlights the similarities between if/else constructs and switch statements in C. In other words, the switch statement in C is simply a specialized and simplified form of the class if/else construct. As you probably know by now, in order to do this problem, you have to understand the basic switch statement syntax for the C programming language.

- As C constructs become more complicated, the compiler will have different options in the implementation of those constructs. Furthermore, different compilers may also implement the same construct in different ways. We mention this here to emphasize that this solution is by no means unique, so there are many different valid solutions to this problem.

- The solution steps through the switch statement implementing a series of if/else constructs. The assembly code solution nicely divides the different options presented in the original C code. Nicely formatted code such as this is a great example of how your assembly code should appear. Specifically, we did not have to provide the extra whitespace in the solution, but doing so makes the solution easier to understand. Additionally, the solution provides a few extra labels that it does not use in a further attempt at making the code more readable and understandable.

- Note that even though there was nothing that required initialization, the code includes an “init” label. This label (and comment) serve as a message to someone reading your code that you did not forget to initialize stuff; there just wasn’t anything that required initialization.

Example 15-12:

Write RAT assembly language code that implements the following C programming construct. Assume r17 holds x_val, r23 holds sensor_23, r24 holds sensor_24, and r11 holds f_val.

```
#define  C_INC        93
#define  RESET_VAL    44
#define  CLAMP_VAL   156


if (x_val <= C_INC + f_val)  {
  sensor_23 = CLAMP_VAL;
}
else  {
  sensor_24 = RESET_VAL;
}
```

Solution: One of the most confusing constructs to create in assembly language is comparisons based on "≤" and "≥". In truth, it's not that hard to do, but it is tricky to do efficiently. This problem outlines "the trick" for one of these comparison operators. As you can see from the result, you can always do the operation efficiently if you spend enough time thinking about it. The first solution is the slightly less efficient approach in that it uses more instructions to do the same thing as the alternate solution.

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU C_INC     = 0x5D   ; chump value
(04)  .EQU RESET_VAL = 0x2C   ; toad value
(05)  .EQU CLAMP_VAL = 0x9C   ; mousy value
(06)  ;-------------------------------------------------------------------------
(07)  init:                             ; nothing to init
(08)
(09)              MOV     r0,C_INC      ; start compare value build
(10)              ADD     r0,r11        ; add f_val
(11)
(12)              CMP     r17,r0        ; do initial comparison
(13)              BREQ    if1           ; branch to "if" if equal
(14)              BRCC    else1         ; branch to "else" if op_left > op_right
(15)  if1:        MOV     r23,CLAMP_VAL ; "if" portion: assign sensor_23
(16)              BRN     done          ; jump over else instructions
(17)
(18)  else1:      MOV     r24,RESET_VAL ; implements the "else" part of C code
(19)
(20)  done:                             ; self-commenting label
(21)  ;-------------------------------------------------------------------------
```

Figure 15-13: Solution for Example 15-10.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- The solution uses .EQU assembler directives for the associated compiler directives (the #defines) in the C code. This is a good choice.
- An equality statement drives the "if" portion of the C code. The solution implements this equality check using a CMP instruction (line 09).

- If the two operands in the CMP instruction are equal, the code branches to the "if" portion of the code, starting at line 14. If the two operands are not equal, program flow drops to line 13 to check to see if r17 is less than r0; in this case the C flag would be set. If the CMP operation does not set the C flag, then the value in r0 is larger than r17 and the code branches to the "else" instruction on line 17.

An important alternate solution for this problem:

```
(00)  ;-------------------------------------------------------------------------
(01)  ;- Assembler Directives
(02)  ;-------------------------------------------------------------------------
(03)  .EQU C_INC     = 0x5D   ; chump value
(04)  .EQU RESET_VAL = 0x2C   ; toad value
(05)  .EQU CLAMP_VAL = 0x9C   ; mousy value
(06)  ;-------------------------------------------------------------------------
(07)  init:                             ; nothing to init
(08)
(09)              MOV     r0,C_INC      ; start compare value build
(10)              ADD     r0,r11        ; add f_val
(11)
(12)              CMP     r0,r17        ; do initial comparison
(13)              BRCS    else1         ; jump to else if op_right < op_left
(14)              MOV     r23,CLAMP_VAL ; "if" portion: assign sensor_23
(15)              BRN     done          ; jump over else instructions
(16)
(17)  else1:      MOV     r24,RESET_VAL ; implements the "else" part of C code
(18)
(19)  done:                             ; self-commenting label
(22)  ;-------------------------------------------------------------------------
```

Figure 15-14: Alternate solution for Example 15-10.

Notes Regarding the Alternate solution: Fun stuff embedded in the alternate solution. The moral of this story is that there is always an "easier" way to implement comparisons based on "≤" and "≥" in assembly language. If you spend a moment with it, you can always figure it out. Note that the "trick" we're referring to is only a matter of switching the operands in the CMP instruction, so it's not that complexicated.

- This solution differs from the previous solution in that it swaps the operands associated with the CMP instruction on line 11. This swaps allows us to use only one branch instruction of the if/else construct (recall that the previous solution used two branch-type instructions). The solution subtracts the expression on the left side of the "<=" from the expression on the right. If this operation results in the C flag being set, the solution branches to the instruction associated "else" on line 16; otherwise, the solution executes the instruction associated with the if on line 13. This is somewhat clever and tricky, but it is equivalent to the previous solution but requires one less instruction.

Example 15-13:

Write RAT assembly language code that implements the following C programming construct. Assume r10 holds acc_val, r11 holds add_val, and r15 holds count.

```
#define  VAL_X   0x77

count = 0;
acc_val = 0;

while (acc_val <  VAL_X)  {
   acc_val += add_val
   count++;
}
```

Solution:

```
(00) ;-------------------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-------------------------------------------------------------------------
(03) .EQU VAL_X     = 0x77   ; random stupid value
(04) ;-------------------------------------------------------------------------
(05)
(06)
(07) init:       MOV     r15,0x00       ; initialize count
(08)             MOV     r10,0x00       ; initialize acc_val
(09)
(10) loop:       CMP     r10,VAL_X      ; compare register to constant
(11)             BRCC    done           ; jump if C=1 (sub generated carry)
(12)             ADD     r10,r11        ; accumulate acc_val variable
(13)             ADD     r15,0x01       ; increment count
(14)             BRN     loop           ; jump to attempt new iteration
(15)
(16) done:                              ; do something else
     ;-------------------------------------------------------------------------
```

Figure 15-15: Solution for Example 15-13.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- It is not known at compile time how many times the loop will iterate because the code bases the while loop stopping condition as a variable that does not have a stated value in the original C code fragment (specifically, the compiler does not know the value of variable "add_val" at compile time). This means the form of the assembly language solution is an iterative construct where the programmer does not know the number of times the loop iterates when he/she/it writes the code; the iteration count is thus variable.

- Once again, the solution uses an .EQU assembler directive for the associated compiler directive (the #defines) in the C code. This is another good choice.

- The way the solution structures the code, the "count" variable stores the number of times the solution iterates the loop and uses the "acc_val" variable to accumulate the result of the additions in the body of the loop. This means that we must first initialize both of these values to zero, which the solution does on lines 06-07.

- This is a while loop; this means the loop may never actually iterate a single time because the solution checks the stopping condition of the while loop before starting the first iteration. The solution does this check on line 09 with a CMP instruction.

- The code exits the loop when the accumulated value (acc_val) exceeds a given threshold (VAL_X). The assembly code indicates this condition by subtracting the threshold value from the accumulated value using a CMP instruction. In this way, the C flag is set when the accumulated value is greater than the threshold value.

Example 15-14:

Write RAT assembly language code that implements the following C programming construct. Assume r10 holds acc_val, r11 holds add_val, and r15 holds count.

```
#define  VAL_X   0x77

count = 0;
acc_val = 0;

do  {
   acc_val += add_val
   count++;
}
while (acc_val <  VAL_X);
```

Solution:

```
(00) ;-------------------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-------------------------------------------------------------------------
(03) .EQU VAL_X     = 0x77   ; random stupid value
(04) ;-------------------------------------------------------------------------
(05)
(06) init:       MOV     r15,0x00       ; initialize count
(07)             MOV     r10,0x00       ; initialize acc_val
(08)
(09) loop:       ADD     r10,r11        ; accumulate acc_val variable
(10)             ADD     r15,0x01       ; increment count
(11)             CMP     r10,VAL_X      ; compare register to threshold
(12)             BRCC    done           ; jump if C=1 (sub generated carry)
(13)             BRN     loop           ; jump to attempt new iteration
(14)
(15) done:                              ; do something else
(16) ;-------------------------------------------------------------------------
```

Figure 15-16: Solution for Example 15-13.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- Yes, this problem is very similar to the previous problem, and because of this, we'll skip some of the details in this problem that we discussed in the previous problem.

- The intention of this problem is two-fold: first, to remind you of the notion of a do-while loop in C, and second, to show the subtle differences between the implementations of a while loop and a do-while loop in assembly language.

- The difference between a do-while and a while loop is that in a while loop, the code checks the ending condition before it executes the body of the loop. In this way, the body of the while loop may never be executed. In contrast, the do-while will always execute the body of the loop at least one time because it always executes the body of the loop at least one time before checking the loop ending condition.

- The assembly code for this solution is the same as the previous solution except we rearrange the solution to ensure that the solution executes the body of the loop one time before it checks the ending condition.

15.4 More Advanced RAT Programming Problems

This section continues with more advanced programming problems. The previous sections provided a basis for many of the standard RAT MCU programming structures. We sincerely hope that after staring at these problems, you did not find them too complicated. Recall that the nice thing about assembly language programming is that nothing can really become too complicated based on the inherently simplistic nature of the assembly language programming.

Example 15-15:

Write a RAT assembly language program that does the following: If SW(3) is pressed [SW(3)='1'], then turn on LED(7) and turn off LED(6) (the other LEDs are not affected); otherwise turn on LED(6) and turn off LED(7) (the other LEDS are not affected). Assume there are eight switches connected to PORT 0x30 and eight LEDs connected to PORT 0x0C. Keep performing these functions forever. Assume that r22 holds the current state of the LEDs.

Solution:

```
(00) ;-----------------------------------------------------------------------
(01) ;- Assembler Directives
(02) ;-----------------------------------------------------------------------
(03) .EQU SW3_MASK       = 0x08   ; switch 3 mask
(04) .EQU LED6_ON_MASK   = 0x40   ; LED 6 mask
(05) .EQU LED6_OFF_MASK  = 0xBF   ; LED 6 mask
(06) .EQU LED7_ON_MASK   = 0x80   ; LED 7 mask
(07) .EQU LED7_OFF_MASK  = 0x7F   ; LED 7 mask
(08) .EQU LED_PORT       = 0x30   ; input port for switches
(09) .EQU SWITCH_PORT    = 0x0C   ; output port for LEDS
(10) ;-----------------------------------------------------------------------
(11) .CSEG                        ; indicates code segment
(12) .ORG   0x01                  ; sets the code segment counter to 0x00
(13)
(14) init:                        ; nothing to init
(15)
(16) main:       CALL    Led_junk
(17)             BRN     main
(18)
(19) ;-----------------------------------------------------------------------
(20) ;- subroutine: Led_junk
(21) ;-
(22) ;- Tweaks with LEDs in specified manner; nothing too exciting.
(23) ;-
(24) ;- Affected regs & flags: r0, r22, C, Z
(25) ;-----------------------------------------------------------------------
(26) Led_junk:
(27) init_1:                              ; nothing to init
(28)             IN      r0,SWITCH_PORT   ; get data from input port
(29)             TEST    r0,SW3_MASK      ; check state of sw3
(30)             BREQ    sw3_nopr         ; jump if switch not pressed (Z=1)
(31)
(32) sw3_press:  AND     r22,LED6_OFF_MASK  ; turn off only LED6
(33)             OR      r22,LED7_ON_MASK   ; turn on only LED7
(34)             OUT     r22,LED_PORT       ; output data to LEDs
(35)             RET                        ; exit
(36)                                        ;
(37) sw3_nopr:   OR      r22,LED6_ON_MASK   ; turn on only LED6
(38)             AND     r22,LED7_OFF_MASK  ; turn off only LED7
(39)             OUT     r22,LED_PORT       ; output data to LEDs
(40)             RET                        ; exit
(41) ;-----------------------------------------------------------------------
```

Figure 15-17: Solution for Example 15-15.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- Although the problem statement did not request it, we provide the solution in the form of the main code continuously calling a subroutine that completes the required functionality. The general form of this solution is thus the "main" code calling individual subroutines. In this case, there is only one subroutine.

- As you can see from the solution, this type of problem utilizes many different "bit mask". When this occurs, the "good" practice is to use an assembler directive (.EQU) to define the masks that the current code is using. Moreover, the customs is to place these assembler directives somewhere before the first line of executable code.

- The bit mask assembler directives use labels that somewhat describe the purpose of the bit mask. This is not always easy. On one hand, you want the assembler directive labels as descriptive as possible, but on the other hand, you don't want the labels too long as they tend to mess up the alignment of text in your source code. The code in this solution does a decent job of providing labels that are not too long but somewhat descriptive as to their purpose in the source code.

- One of the underlying goals of this program is only to change the bits specifically mentioned in the problem description. Note that the bit masks only affect the bits in question in the give register thus protecting all of the other bits. In this context, the notion of "protecting" means that we are purposely not changing the value of those bits as we change the other bits (as directed by the problem).

- The solution opts to allow the value in r0 to be lost each time the program calls this subroutine. We could have pushed this value to protect it, but it was not in the cards for this problem.

Example 15-16:

Write a RAT assembly language subroutine that sorts three values. The three values are store in register r20, r21, and r22.

Solution: The key to solving this problem in an orderly manner is to recall that Example 15-5 involved a two-value sorting problem (we'll not include that solution again). The final solution to Example 15-5 was a subroutine that sorted two values, which proves to be rather useful in this problem for those of us who don't want to reinvent the wheel with their solution. Thus, the solution to this problem advertises the notion of code reuse. Check back on the solution to Example 15-5 for further details.

Additionally, sorting is a common practice out there in computer programming-land. That being the case, there are many different sorting algorithms out there that we could employ. The solution to this problem uses what we refer to as a "bubble sort", which not the most efficient but is probably the easiest to understand.

```
(00) ;-----------------------------------------------------------------------
(01) ;- subroutine: Sort_3
(02) ;-
(03) ;- Performs a sort on three values sent in r20, r21, & r22. The larger of the
(04) ;- two numbers is placed in r22; the smaller in r20.
(05) ;-
(06) ;- Affected regs & flags: maybe r20, r21, r22, C, Z
(07) ;-----------------------------------------------------------------------
(08)
(09) Sort_3:
(10) init:                              ; nothing to init
(11)             MOV     r31,r20        ; load registers for SR call
(12)             MOV     r30,r21
(13)             CALL    Sort_2         ; call 2-value sort subroutine
(14)             MOV     r21,r31        ; adjust regs for sorted results
(15)             MOV     r20,r30
(16)
(17)             MOV     r31,r22
(18)             MOV     r30,r21
(19)             CALL    Sort_2         ; call 2-value sort subroutine
(20)             MOV     r22,r31        ; adjust regs for sorted results
(21)             MOV     r21,r30
(22)
(23)             MOV     r31,r20        ; load registers for SR call
(24)             MOV     r30,r21
(25)             CALL    Sort_2         ; call 2-value sort subroutine
(26)             MOV     r21,r31        ; adjust regs for sorted results
(27)             MOV     r20,r30
(28)
(29)             RET                    ; done at last
(30) ;-----------------------------------------------------------------------
```

Figure 15-18: Solution for Example 15-16.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- There is only one detail regarding the bubble sort that you need to know: the worst case for the given number of items you need to sort. In the worst case, the largest value may be on the bottom, so it will then need to "bubble up" to the top. This means that we need to do at least two "iterations" of the bubble sort in the solution. The solution implements these two iterations in a smart manner. If first bubbles the largest value to the top (the first two calls to the Sort_2) and then it sorts the two lower values. This approach is not generic but it works.

- The problem with this example is that it lacks the ability to be generic in that it stores the values that need sorting in registers. In this case, some type of generic subroutine would be better in the case that we sort a greater number of values. A later example performs a sort on values stored in scratch RAM, which proves to be a more generic approach.

Example 15-17:

Write a RAT assembly language subroutine that does the following. This subroutine compares two values sent in r10, and r11. If the two values are equivalent, both r10 and r11 are set to 0x00. If r10 is greater than r11, then r10 is set 0x88 and r11 is set to 0x77. If r11 is greater than r10, then both r10 and r11 are set to 0xFF.

Solution: This problem highlights a set of comparison operations that you may need to do.

```
(00) ;-------------------------------------------------------------------
(01) ;- subroutine: Comp_reg
(02) ;-
(03) ;- Compares two register: r10 & r11, the does various pointless items
(04) ;- based on results of comparisons.
(05) ;-
(06) ;- Affected regs & flags: maybe r20, r21, r22, C, Z
(07) ;-------------------------------------------------------------------
(08)
(09) Comp_reg:
(10) init:                              ; nothing to init
(11)             CMP     r10,r11        ; compare registers
(12)             BREQ    equal          ; they are equal
(13)             BRCS    less_than      ; r10 less than r11
(14)
(15) great_than: MOV     r10,0x88       ; r10 > r11 case
(16)             MOV     r11,0x77       ; perform required assignments
(17)             RET
(18)
(19) equal:      MOV     r10,0x00       ; r10 = r11 case
(20)             MOV     r11,0x00       ; perform required assignments
(21)             RET
(22)
(23) less_than:  MOV     r10,0xFF       ; r10 < r11 case
(24)             MOV     r11,0xFF       ; perform required assignments
(25)             RET
     ;-------------------------------------------------------------------
```

Figure 15-19: Solution for Example 15-17.

Notes Regarding the Solution: Fun stuff embedded in the solution.

The form of the solution takes one of three paths. The solution first checks for equality and then checks to see if r10 is less than r11. If these, two checks both fail, then r10 must be greater than r11. The point here is that

there are many different but equivalent solutions to this problem; please don't get stuck thinking this is the only possible solution (although this is a rather clean set of code).

Lines (10-11) contain something that you should always be on the lookout for in assembly language program. Often times when you have a conditional branch statement following another conditional branch statement, you can often assume the programmer did not know what they were doing. However, this is not the case for this solution. The reason this is OK in this solution is that the CMP instruction sets the condition flags and the conditional branch instruction do not change them. If the two conditional branch instructions fail, the code drops down to the final condition in the solution. Somewhat tricky, but something you'll see and use quite often in assembly language programming.

The solution contain three RET instructions. While it would be possible to write this solution using unconditional branches and using only one RET instruction, you could not write this solution using less instructions. The better choice is to have three RET instructions, which makes the code more clear. Additionally, subroutines typically have one entry point and many exit points.

Example 15-18:

Write a RAT assembly language subroutine that swaps the lower and higher order nibbles in r8. Don't use more than 11 instructions in your solution.

Solution: Tweaking bits is something you do often when dealing embedded systems that actually control something. The notion in this problem is that the individual bits may be controlling something of interest, so let's pretend they do for this problem.

```
(00) ;-----------------------------------------------------------------------
(01) ;- subroutine: Nib_swap
(02) ;-
(03) ;- Exchanges the higher and lower order nibbles in r8.
(04) ;-
(05) ;- Affected regs & flags: r8, r10, r4, r9, C, Z
(06) ;-----------------------------------------------------------------------
(07)
(08) Nib_swap:
(09) init:       MOV     r9,r8          ; copy value
(10)             MOV     r4,0x04        ; load iterative count
(11)
(12) loop:       ROL     r9             ; rotate both nibs 4x
(13)             ROR     r8
(14)             SUB     r4,0x01        ; decrement loop count
(15)             BRNE    loop
(16)
(17)             AND     r9,0xF0        ; clear bottom nibble
(18)             AND     r8,0x0F        ; clear upper nibble
(19)             OR      r8,r9          ; combine nibble results
(20)
(21)             RET                    ; take it on home
(22) ;-----------------------------------------------------------------------
```

Figure 15-20: Solution for Example 15-18.

Notes Regarding the Solution: Fun stuff embedded in the solution. Once again, you can solve this problem in many ways; there is nothing too special about this solution. One key in the problem is to limit the number of instructions that you use in order to force you to be clever. Please recall that a nibble refers to a 4-bit value.

- This solution highlights bit-manipulation such as rotating and masking. As you'll soon see, there is a better solution for this problem. What you should gather from this solution is the structure and organization of the solution as you'll do this type of stuff often.

- The approach here is rather standard for bit manipulation: save the value (line 09), load an iterative counter (line 10), do the shifting (lines 12-15), clean up the values (lines 17-18), and massage the result (line 19).

- Lines 17-18 use masks; it would be better to use .EQU to define these masks rather than the actual numerical values. We omitted them here because we generally place assembler directives at the beginning of the program; the solution to this problem is a subroutine.

Figure 15-21 and Figure 15-22 show two alternative solutions to this example, with both solutions being more efficient than the first solution. As you can see, you can swap nibbles using only rotate-type instructions. This being the case, you have two options: use an iterative loop or use as many rotate instructions as you need to get the job done.

We include these two solutions for a reason. As you can see with this example, using an iterative loop uses the same number of instructions as not using a loop. These solutions have the same number of instructions because you had to do something four times. If you had to "do something" five times, using an iterative construct is more efficient. If you had to "do something" three times, writing the instructions out not using a loop would be more efficient because you don't have to deal with the loop overhead. The notion of loop overhead involves the setting of the iterative count, the decrementing of the iterative value, and the conditional branch instruction. While all this seems somewhat nitpicky, you generally always need to consider whether dealing with the loop overhead is actually worthy it in terms of overall efficiency of the code.

```
(00) ;-----------------------------------------------------------------------
(01) ;- subroutine: Nib_swap
(02) ;-
(03) ;- Exchanges the higher and lower order nibbles in r8.
(04) ;-
(05) ;- Affected regs & flags: r8, r10, r4, r9, C, Z
(06) ;-----------------------------------------------------------------------
(07)
(08) Nib_swap:
(09) init:       MOV     r4,0x04        ; load iterative count
(10)
(11) loop:       ROL     r8             ; rotate the value 4x
(12)             SUB     r4,0x01        ; decrement loop count
(13)             BRNE    loop
(14)
(15)             RET                    ; take it on home
(16) ;-----------------------------------------------------------------------
```

Figure 15-21: An alternative solution for Example 15-18.

```
(00) ;-----------------------------------------------------------------
(01) ;- subroutine: Nib_swap
(02) ;-
(03) ;- Exchanges the higher and lower order nibbles in r8.
(04) ;-
(05) ;- Affected regs & flags: r8, r10, r4, r9, C, Z
(06) ;-----------------------------------------------------------------
(07)
(08) Nib_swap:
(09) init:                          ; nothing to init
(10)             ROR     r8         ; rotate the value 4x
(11)             ROR     r8
(12)             ROR     r8
(13)             ROR     r8
(14)
(15)             RET                ; take it on home
(16) ;-----------------------------------------------------------------
```

Figure 15-22: Yet another alternative solution for Example 15-18.

Example 15-19:

Write a RAT assembly language subroutine that multiples the value in r15 by 32. The final result will potentially be greater than 255, so the result of this multiply operation should be stored in r15 and r16, with r16 being the higher-order byte. Don't use more than 11 instructions in your solution.

Solution:

```
(00) ;-----------------------------------------------------------------
(01) ;- subroutine: Mult_32
(02) ;-
(03) ;- Multiplies the number in r15 by 32. The result is stored in
(04) ;- r15 (lower-order byte) and r16 (higher-order byte).
(05) ;-
(06) ;- Affected regs & flags: r4, r15, r16, C, Z
(07) ;-----------------------------------------------------------------
(08) .EQU  B0_1_MASK = 0x01
(09) .EQU  B0_0_MASK = 0xFE
(10)
(11) Mult_32:
(12) init:       MOV     r16,0x00       ; clear higher order byte
(13)             MOV     r4,0x05        ; load iterative count: 2^5=32
(14)
(15) loop:       LSL     r16            ; shift both bytes left
(16)             LSL     r15            ;
(17)             BRCC    no_car         ; handle no carry
(18)             OR      r16,B0_1_MASK  ; set b(0) in higher-order byte
(19)
(20)             BRN     repeat         ; jump over clear-bit operation
(21) no_car:     AND     r16,B0_0_MASK  ; clear b(0) in higher-order byte
(22)
(23) repeat:     SUB     r4,0x01        ; decrement loop count
(24)             BRNE    loop           ; are we done yet?
(25)
(26)             RET                    ; It's done Jimmy
(27) ;-----------------------------------------------------------------
```

Figure 15-23: Solution for Example 15-19.

Notes Regarding the Solution: Fun stuff embedded in the solution. Once again, the problem statement limits the number of instructions in your solution in order to encourage you to generate a clever and efficient solution. As with a previous problem, you can be clever with this solution; we'll provide two solutions so highlight some typical bit manipulation operations before we show you the even more clever solution. Here are some other fun things in the solution.

- The main issue with the problem is that you have an 8-bit number that you're multiplying by 32, which means you could have a 13-bit result. This means that you'll need two registers for this solution, and some clever shift operations along the way.

- We're limited to not more than 11 instructions, so we'll use iteration in the solution. This requires that we set the loop count in the beginning of the subroutine as part of the initialization portion of the subroutine (line 13). The other part of the initialization is line 12 where we clear the register we'll be using as the higher-order byte.

- Recall that you can perform a multiply by 32 by shifting the value left five times and inserting zeros on the right side of the value. Since we're working with two register in this problem, we always need to shift both registers (we'll never shift just one). Lines (15-16) show the left shift operations.

- The idea with this solution is that we do a left-shift on the lower-order byte and check the carry flag. If the carry flag is set, meaning we shift out a '1' from bit-7 of the lower-order byte, then we manually set the LSB of the higher-order byte using an OR operation. If the carry flag is cleared, we manually clear the LSB of the higher-order byte using an AND operation. We do this five times.

- This problem opts to use .EQU directives and include them in the subroutine body. It would be better to places all .EQU directives at the beginning of the source code rather than in subroutine bodies.

This solution is all fine and dandy, but Figure 15-24 shows the easier solution. This solution takes advantage of the fact that the LSL instruction shifts the MSB into the carry and the carry into the LSB. In this way, we don't need to check on the value of the carry; we simply shift it into the LSB of the higher-order byte. Rather clever indeed; this is one of the many tricks you try to apply as it saves four instructions over this problem's first solution. Saving four instructions make the subroutine run significantly faster. Keep in mind that you must shift the lower-order byte first in order for this solution to work.

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- subroutine: Mult_32
(02)  ;-
(03)  ;- Multiplies the number in r15 by 32. The result is stored in
(04)  ;- r15 (lower-order byte) and r16 (higher-order byte).
(05)  ;-
(06)  ;- Affected regs & flags: r4, r15, r16, C, Z
(07)  ;-----------------------------------------------------------------------
(08)
(09)  Mult_32:
(10)  init:       MOV     r16,0x00        ; clear higher order byte
(11)              MOV     r4,0x05         ; load iterative count: 2^5=32
(12)
(13)  loop:       LSL     r15             ; shift both bytes left
(14)              LSL     r16             ;
(15)
(16)              SUB     r4,0x01         ; decrement loop count
(17)              BRNE    loop            ; are we done yet?
(18)
(19)              RET                     ; It's done Jimmy
(20)  ;-----------------------------------------------------------------------
```

Figure 15-24: Solution for Example 15-19.

Example 15-20:

Write a RAT assembly language subroutine that counts the number of bits that are set in r30 and stores the value in r31. The subroutine should not alter the contents of r30. Don't use more than ten instructions in your solution.

Solution:

```
(00) ;--------------------------------------------------------------------
(01) ;- subroutine: Bit_count
(02) ;-
(03) ;- This subroutine counts the number of set bits in r30 and places
(04) ;- value in r31. ;-
(05) ;-
(06) ;- Affected regs & flags: r0, r31, C, Z
(07) ;--------------------------------------------------------------------
(08)
(09) Bit_count:
(10) init:       MOV     r31,0x00       ; set bit count at 0
(11)             PUSH    r1             ; save r1 value
(12)             MOV     r1,0x08        ; load iterative count
(13)
(14) loop:       ROR     r0             ; shift LSB into C
(15)             BRCC    incr           ; check carry, branch if not set
(16)             ADD     r31,0x01       ; increment bit count
(17)
(18) incr:       SUB     r1,0x01        ; decrement loop count
(19)             BRNE    loop           ; continue with loop
(20)
(21)             POP     r1             ; restore original r1 value
(22)             RET                    ; take it on home
(23) ;--------------------------------------------------------------------
```

Figure 15-25: Solution for Example 15-20.

Notes Regarding the Solution: Fun stuff embedded in the solution. This is another problem that involves clever bit manipulation, which means you need to completely understand the ramifications of all the bit-crunching instructions. Here are some other interesting items regarding the solution.

- This solution is one of many possible solutions. The two main approaches to solving this problem are using bit-masking or using the carry flag. This solution opts to use the carry flag.
- The solution saves the r1 register by pushing it onto the stack at the beginning of the subroutine and popping it off the stack before the subroutine exits.
- There is no need to save the original value of r0 because after we shift r0 eight times, the value in r0 will be the same as it was when we entered the subroutine.
- The general form of the solution is an iterative loop with one iteration per bit in a RAT register. The solution uses a LSR instruction on line (15); this instruction shifts the LSB of the register into the carry flag. If the carry flag is set, the solution increments a count register; otherwise, the code iterates until the loop count runs out.

Example 15-21:

Write a RAT assembly language subroutine that swaps the ordering of bytes in r22. Specifically, bit(0) should become bit(7), bit(1) should be bit(6), etc. Don't use more than ten instructions in your solution.

Solution:

```
(00) ;-----------------------------------------------------------------
(01) ;- subroutine: Bit_swap
(02) ;-
(03) ;- This subroutine swaps the order of bits in r22.
(04) ;-
(05) ;- Affected regs & flags: r20, C, Z
(06) ;-----------------------------------------------------------------
(07)
(08) Bit_swap:
(09) init:       PUSH    r1              ; save r1 value
(10)             MOV     r20,r22         ; copy value
(11)             MOV     r1,0x08         ; load iterative count
(12)
(13) loop:       LSR     r20             ; shift LSB into C
(14)             LSL     r22             ; check carry, branch if not set
(15)
(16)             SUB     r1,0x01         ; decrement loop count
(17)             BRNE    loop            ; continue with loop
(18)                                     ;
(19)             POP     r1              ; restore r1 value
(20)             RET                     ; take it on home
(21) ;-----------------------------------------------------------------
```

Figure 15-26: Solution for Example 15-21.

Notes Regarding the Solution: Fun stuff embedded in the solution. This is yet another bit-manipulation problem. From my experience giving this problem to people, they seem to want to use some type of bit-masking to arrive at the solution. While you can do it this way, the solution would be long and confusing. Here are the interesting comments regarding the solution.

- The general form of this solution is that we need to do something for each bit location in the RAT register, which is of course eight. This requires an iterative loop with an iterative count of eight. Wow!

- We use the value of r1 for the iterative count, so we opt to save this value by pushing on the stack at the beginning of the subroutine and popping it off once the subroutine exits. Additionally, we save the original value by copying it a working register on line (10).

- This solution is clever and requires intimate understanding of the LSR and LSL instruction. The LSR instruction on line 13 shifts the LSB into the carry flag. The following LSL instruction shifts that carry flag into the LSB of the new register. In this way, the LSB of the r20 value starts filling the r22 value starting from the LSB end and working its way to the MSB end of r22. After eight iterations, the value in the original r20 register does indeed have the bits swapped.

Example 15-22:

Write a RAT assembly language subroutine that counts that examines and possibly modifies a value in scratch RAM. The scratch RAM location in question is represented by the value in r20 (the value in r20 is interpreted as an address). If the value at that location is even, then the number is multiplied by four and stored back at that scratch RAM address; otherwise that value at that location is divided by two and stored back at that scratch RAM address. Don't worry about overflow and underflow for this problem.

Solution:

```
(00) ;-----------------------------------------------------------------------
(01) ;- subroutine: mem_op1
(02) ;-
(03) ;- This subroutine examines the memory location specified by r20. If
(04) ;- the value at that location is even, it is multiplied by 4 and
(05) ;- and stored back at the same location; otherwise, the value at that
(06) ;- location is divided by 2 and stored back at that same location.
(07) ;-
(08) ;- Affected regs & flags: C, Z
(09) ;-----------------------------------------------------------------------
(10) .EQU   ODD_EVEN_MASK  = 0x01
(11)
(12) mem_op1:
(13) init:       PUSH    r0                ; save r0 contents
(14)             CLC                       ; pre-clear carry flag
(15)
(16)             LD      r0,(r20)          ; see if user sent zero value
(17)             TEST    r0,ODD_EVEN_MASK  ; check LSB
(18)             BREQ    even              ; branch if even
(19)                                       ;
(20) odd:        LSR     r0                ; divide by 2
(21)             BRN     store             ; finish up
(22)
(23) even:       LSL     r0                ; multiply by 4
(24)             LSL     r0                ; store data in new location
(25)                                       ;
(26) store:      ST      r0,(r20)          ; decrement loop count
(27)             POP     r0                ; restore r0 contents
(28)
(29)             RET                       ; take it on home
(30) ;-----------------------------------------------------------------------
```

Figure 15-27: Solution for Example 15-22.

Notes Regarding the Solution: Fun stuff embedded in the solution. The main purpose of this problem is to highlight the use of indirection in memory references. Also included are binary math by shifting and an if/else construct. Here are some other items regarding the solution. Fun stuff indeed.

- The subroutine officially does not change any register as r0 is pushed/popped upon entering and leaving the subroutine, respectively.
- Any time you're working with shift operations, you generally must place the carry flag in a known state. The solution does this by clearing the carry flag on line (14).
- The subroutine loads the contents of memory into an arbitrary working register, r0 (line 16). The problem description states that you can find the memory address in question in register r20. This being the case, we must use the indirect LD instruction, which we do on line (16).
- The solution uses left shifts for multiplication and right shift for division, which we refer to as "binary math". Because we pre-set the carry flag to zero, we guarantee ourselves clean results as the

carry flag shifts into the register as part of the LSL and LSR instructions. In order to make the problem simpler, we opt to ignore overflow and underflow.

Example 15-23:

Write a RAT assembly language subroutine that implements a C memcpy() function. See the C definition for a memcpy() below. For this subroutine, assume that s1 is provided in r1 and s2 is provided in r2, respectively; the value of n is provided in r10. Your function should copy n-bytes of data starting at the scratch RAM location specified in r1 to the scratch RAM locations specified in r2. For this problem, you can assume the n-bytes do not exceed the upper or lower boundary of the scratch RAM.

```
memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

The *memcpy*() function shall copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*.

Solution:

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- subroutine: memcpy
(02)  ;-
(03)  ;- This subroutine copy n-bytes of date (n is sent in r10) from the
(04)  ;- scratch RAM locations starting at the address value in r2 to the
(05)  ;- scratch RAM locations start at the value in r1.
(06)  ;-
(07)  ;- Affected regs & flags:: r10, C, Z
(08)  ;-----------------------------------------------------------------------
(09)
(10)  Memcpy:     CMP     r10,0x00        ; see if user sent zero value
(11)              BRNE    do_work         ;
(12)              RET                     ; return to the living
(13)
(14)  init:       PUSH    r0              ; save regs used by subroutine
(15)              PUSH    r1
(16)              PUSH    r2
(17)
(18)  do_work:    LD      r0,(r2)         ; get data to be copied
(19)              ST      r0,(r1)         ; store data in new location
(20)
(21)              ADD     r2,0x01         ; increment r2 pointer
(22)              ADD     r1,0x01         ; increment r1 pointer
(23)
(24)              SUB     r10,0x01        ; decrement loop count
(25)              BRNE    do_work         ; continue with loop
(26)
(27)  restore:    POP     r2              ; restore saved registers
(28)              POP     r1
(29)              POP     r0
(30)
(31)              RET                     ; take it on home
(32)  ;-----------------------------------------------------------------------
```

Figure 15-28: Solution for Example 15-23.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- As with many subroutines that do something a certain number of times, you should check to see that the number of times you need to do something is non-zero. This is what this subroutine does on line 10. This is good practice as if you don't do the check, your subroutine may fail in a bad way depending upon how you model the iterative part of the subroutine.

- This subroutine save three of the registers it uses (lines 14-16) but opts to not save the register that holds the iterative count, which is the number of lines of data that need copying. This is arbitrary; no big secret here. The solution restores the saved registers before exiting the subroutine (line 27-29).

- The notion of indirection makes this solution relatively simple. The solution copies the data in question from one memory location to a working register (line 18), and then from the working register to another memory location (line 19). You gotta love that indirection. Where would we be without it?

Example 15-24:

Write a RAT assembly language subroutine that divides a number in register r10 by three. Place the quotient in r11 and the remainder in r12.

Solution:

```
(00)  ;-------------------------------------------------------------
(01)  ;- subroutine: Divby3
(02)  ;-
(03)  ;- This subroutine divides the value in register r10 by three.
(04)  ;- The quotient is placed in register r11 and the remainder is
(05)  ;- placed in register r12. The algorithm does division by
(06)  ;- iterative subtraction. When the division is an even
(07)  ;- multiple of the number, the subtraction will eventually
(08)  ;- result in the zero flag being set on the final iteration.
(09)  ;- Negative results will set the carry flag which indicates
(10)  ;- the divisor did not divide evenly into the number.
(11)  ;-
(12)  ;- Registers tweaked: r10,r11,r12,r2
(13)  ;-------------------------------------------------------------
(14)  Divby3:
(15)  init:       PUSH    r2            ; save state of r2
(16)              MOV     r11,0x00      ; init quotient
(17)                                    ;
(18)  loop:       MOV     r2,r10        ; copy divisor to r2
(19)              SUB     r10,0x03      ; subtract divisor
(20)              BRNE    not_zero      ; jumps if result not zero
(21)                                    ;
(22)  divsr_zero: MOV     r12,0x00      ; load zero remainder
(23)              ADD     r11,0x01      ; increment r11 (quotient)
(24)              RET                   ; all done
(25)                                    ;
(26)  not_zero:   BRCS    neg_val       ; branch if subtract was negative
(27)              ADD     r11,0x01      ; increment r11 (quotient)
(28)              BRN     loop          ; iterate again
(29)                                    ;
(30)  neg_val:    MOV     r12,r2        ; r2 has the remainder
(31)              POP     r2            ; restore state of r2
(32)              RET                   ; all done
(33)  ; -------------------------------------------------------------
```

Figure 15-29: Solution for Example 15-24.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- You've probably noticed that the RAT MCU does not have an instruction that does division. The alternative is to do subtractions by the divisor while keeping track of how many subtractions you do. Eventually you'll end up either at zero or less than zero from one of the subtractions. The general form of the code is to do the subtraction iteratively while checking the value of each subtraction result.

- The solution uses register r2 as a working register, so it saves and restores r2 at the beginning and end of the subroutine, respectively.

- If the value being divided is non-zero after the subtraction, the result indicates there are more iterations to do (the greater than zero case) or that the algorithm is completed. When the algorithm is complete, the solution copies the result of the subtraction to r12, as this is not the remainder, and exits the subroutine. The solution increments the quotient in cases where the subtraction result is zero (line 26) or non-negative (line 59).

Example 15-25:

Using the LUT information provided below, write a RAT assembly language subroutine that calculates the parity of the value in r20. Indicate odd or even parity by setting or clearing the C, respectively.

- Do not permanently change any register value
- Minimize the number of instructions in your solution

```
DSEG
.ORG     0x80

         ; number of bits set in each unique four-bit value
         ;  in the range: [0,15])

.DB      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03
.DB      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04
```

Solution:

```
(00)  ;-------------------------------------------------------------
(01)  ;- subroutine: Parity_tab
(02)  ;-
(03)  ;- This subroutine uses a LUT to calculate the parity of the
(04)  ;- value in r20. This subroutine returns parity with the
(05)  ;- C flag with '1' and '0' indicating odd and even parity,
(06)  ;- respectively.
(07)  ;-
(08)  ;- Registers tweaked: none
(09)  ;-------------------------------------------------------------
(10)  Parity_tab:
(11)  init:       PUSH    r20           ; save state of r20: low nib
(12)              PUSH    r21
(13)              PUSH    r0
(14)              MOV     r21,r20       ; copy r20 to r21: hi nib
(15)                                    ;
(16)  massage:    AND     r20,0x0F      ; clear hi nib
(17)              ROL     r21           ; swap nibs with 4 rotates
(18)              ROL     r21
(19)              ROL     r21
(20)              ROL     r21
(21)              AND     r21,0x0F      ; clear hi nib
(22)
(23)  offset:     ADD     r20,0x80      ; offset to LUT
(24)              ADD     r21,0x80      ; offset to LUT
(25)                                    ;
(26)  load:       LD      r0,(r20)      ; load zero remainder
(27)              LD      r20,(r21)     ; reuse r21 as temp reg
(28)  sum:        ADD     r0,r20        ; add two halves
(29)              LSR     r0            ; shift LSB to C flag
(30)
(31)              POP     r0            ; restore state
(32)              POP     r21
(33)              POP     r22
(34)
(35)              RET                   ; all done – homeward ho
(36)  ; -------------------------------------------------------------
```

Figure 15-30: Solution for Example 15-24.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- This is the typical parity problem, but the solution uses the LUT shown in the problem description.
- The initialization starting at line (11) include save registers to the stack and making a copy of the sent data in r20.
- The "massage" part of the solution starting at line (16) involves creating two 4-bit values in two different registers; these 4-bit values are the two nibbles in the sent register.
- Because the LUT starts at address 0x80, we need to offset the nibble values before we use them to access the LUT. We do this starting at line (23) at the "offset" label. We then need to access the LUT using the offset nibble data with two LD instructions. We reuse the r20 register on line (27) because we are done with the data in this register; this saves up from having to save and restore another register.
- We add our LUT values on line (28); a right-shift places the LSB into the C flag. Recall that at this point, the LSB represents the oddness or evenness of the parity calculation.
- The state of the MCU before the subroutine call is restore on lines (31-33).

Example 15-26:

Write a RAT assembly language subroutine that inputs a value from port 0xCC, transforms the value according to the following equation: out=1.5(in) + 7, and outputs the result to port 0xCD. Round up the 1.5(in) value (from divide by two operation). Make sure the range of the output value falls within: [19,252].

Solution:

```
(00) ;-------------------------------------------------------------
(01) ;- subroutine: Xform_num
(02) ;-
(03) ;- This subroutine inputs a value from port CC and transforms
(04) ;- the number using the following equation: out=1.5(in) + 7.
(05) ;- The range of output values is forced to be in the range of
(06) ;- [19,252]. The result is written to port CD.
(07) ;-
(08) ;- Affected regs & flags: r0, r1, C, Z
(09) ;-------------------------------------------------------------
(10) ;- Constant declarations:
(11) ;-------------------------------------------------------------
(12) .EQU  HI_MAX    =  0xFC           ; max output val=252
(13) .EQU  LO_MIN    =  0x13           ; min output val=19
(14) ;-------------------------------------------------------------
(15) Xform_num:
(16) Init:                             ; nothing to init
(17)
(18)             IN     r0,0xCC        ; get value to transform
(19)             MOV    r1,r0          ; copy original data
(20)             CLC                   ; clear carry before shift
(21)             LSR    r1             ; divide by two
(22)             ADDC   r0,r1          ; completes 1.5x portion
(23)                                   ;
(24)             ;---------------------------------------------
(25)             ;- test for 8-bit range exceeded
(26)             ;---------------------------------------------
(27) hi_test:    BRCS   hi_clamp       ; number exceeds 8-bit range
(28)             ADD    r0,0x07        ; finish transform
(29)             BRCS   hi_clamp       ; number exceeds 8-bit range
(30)                                   ;
(31)             ;---------------------------------------------
(32)             ;- test for high range exceeded
(33)             ;---------------------------------------------
(34)             CMP    r0,HI_MAX      ; see if result > max value
(35)             BRCC   hi_clamp       ; exceeds upper limit
(36)                                   ;
(37)             ;---------------------------------------------
(38)             ;- test for low range requirements not met
(39)             ;---------------------------------------------
(40) lo_test:    CMP    r0,LO_MIN      ; subtract the lower value
(41)             BRCS   lo_clamp       ; jump low range not met
(42)                                   ;
(43)             BRN    out_val        ; the result must be OK
(44)                                   ;
(45) hi_clamp:   MOV    r0,HI_MAX      ; puts value at HI_MAX
(46)             BRN    out_val        ;
(47)                                   ;
(48) lo_clamp:   MOV    r0,LO_MIN      ; puts value at LO_MIN
(49)                                   ;
(50) out_val:    OUT    r0,0xCD        ; output r0
(51)             RET                   ; all done
(52) ;-------------------------------------------------------------
(53)
(54)
(55)
(56)
(57)
(58)
(59)
```

Figure 15-31: Solution for Example 15-26.

Notes Regarding the Solution: Fun stuff embedded in the solution.

This problem contains a lot of busy work, but some rather common assembly language practices as well. Once you break this problem down into small pieces of functionality, it's not that bad. Though we use microcontrollers primarily to "control things" (which is does by flipping bits), they occasionally need to do some simple math as well. This problem uses assembly language to complete a simple transform.

- The first part of the solutions (lines 18-21) does the math required by the problem. The trick of creating 1.5x with round up involves copying the inputted data, dividing the copy by two, and adding the divided value back to the original value. The solution uses a right-shift for the divide by two (line 18), but nicely clears the carry flag before the actual shift (don't ever forget this). The left-shift places the LSB into the carry; if this carry is set, then the result is should be rounded up, which the solution does with the ADDC instruction on line 22.

- The solution checks the result for a carry on lines 27 and 29. The first check is for the initial result, while the second check is for the addition of seven to complete the transform. If either check indicates a carry, the solution branches to the high clamping code starting on line 45.

- If the transform does not require a high clamp, the solution then checks to see if it requires a low clamping starting on line 40.

- The solution arbitrarily opts not to save either working register (r0 or r1). It would have been better to save and restore these values as part of the subroutine, but we omitted these operations in order to save writing and paper.

Example 15-27:

Write a RAT assembly language subroutine that adds two 2-digit BCD numbers and provides a 3-digit BCD result. The two 2-digit BCD numbers are initially in registers r10 and r11, where the upper and lower nibbles represent the 10's and 1's digit, respectively. The result should be placed in register r20 (the 100's digit in the lower nibble) and r21 (the 10's and 1's digit in upper and lower nibble, respectively). Make sure your subroutine does not alter the contents of either r10 or r11. You can assume the contents of r10 and r11 both contain valid BCD numbers.

Solution:

```
(00) ;-----------------------------------------------------------
(01) ;- subroutine: Add_bcd
(02) ;-
(03) ;- This subroutine adds two 2-digit BCD numbers residing in
(04) ;- r10 & r11. The upper and lower nibble represent the 10's
(05) ;- 1's digit, respectively. The result is placed in r20 for
(06) ;- the 100's digit (lower nibble) and r21 (upper nibble for
(07) ;- the 10's digit and lower nibble for the 1's digit). This
(08) ;- subroutine does not check for valid BCD numbers in r10
(09) ;- & r11.
(10) ;-
(11) ;- Affected regs & flags:: r20, r21, r28, r29, r30, r31, C, Z
(12) ;-----------------------------------------------------------
(13) .EQU LO_NIB_MASK    = 0x0F
(14) .EQU HI_NIB_MASK    = 0xF0
(15) .EQU TENS_INC_VAL   = 0x10
(16) .EQU LO_TEN         = 0x0A
(17) .EQU HI_TEN         = 0xA0
(18)
(19) Add_bcd:
(20) init:       MOV     r28,r10          ; copy input values
(21)             MOV     r29,r11          ;
(22)             MOV     r30,r10          ; copy input values
(23)             MOV     r31,r11          ;
(24)
(25) mask_vals:  AND     r28,LO_NIB_MASK  ; massage nibbles of copied data
(26)             AND     r29,LO_NIB_MASK
(27)             AND     r30,HI_NIB_MASK
(28)             AND     r31,HI_NIB_MASK
(29)
(30) add_ones:   ADD     r28,r29          ; do 1's digit add
(31)             CMP     r28,LO_TEN       ; see if it exceeded 10
(32)             BRCS    lt_10x           ; branch if is did not
(33)             ADD     r30,TENS_INC_VAL ; increment 10's data
(34)             SUB     r28,LO_TEN       ; subtract 10 from 1's digit
(35)
(36) lt_10x:     ADD     r30,r31          ; add the 10's digit
(37)             BRCC    gt_10            ; result greater than 16
(38)             ADD     r30,0x06         ; 16-10
(39)             MOV     r20,0x01         ; increment the 100's spot
(40)             BRN     done             ; increment 100’s
(41)
(42) gt_10:      CMP     r30,HI_TEN       ; see if it exceeded 10
(43)             BRCC    lt_10y           ; no carry; clear 100’s
(44) add_hun:    MOV     r20,0x01         ; increment the 100's spot
(45)             SUB     r30,HI_TEN       ; subtract a up nibble 10
(46)             BRN     done
(47)
(48) lt_10y:     MOV     r20,0x00         ; clear the 100's nibble
(49) done:       MOV     r21,r28          ; pack the upper and lower
(50)             OR      r21,r30          ;  nibble
(51)             RET
(52) ; -----------------------------------------------------------
```

Figure 15-32: Solution for Example 15-27.

Notes Regarding the Solution: Fun stuff embedded in the solution.

- Because the problem requests that you don’t alter the values of the two two-digit BCD numbers, the first task of this subroutine is to save those values by copying them to four other registers. We’ll need both upper and lower nibbles in this problem, which is why we are making four copies rather than two copies.

- The four lines of code starting at the “mask_vals” labels tweak the two BCD numbers in order to prepare them for the nibble-based operations required by this problem. There are basic four-bit masks, which were preset using assembler directives at some other point in the problem.

- The next task starting at the "add_ones" label is to add the two lower nibbles of the input data, which is not surprisingly, the 1's position of the two BCD numbers. The solution then issues a CMP operation in order to see if the result of the addition exceeded a value of ten. If CMP instruction subtracts ten from the result; if this operation causes a carry, then the original addition did not exceed ten (confusing, yes). When there is no carry from the CMP operation, the result of the ADD did exceed ten and a value of one must be added to one of the ten's nibble operations. In this case, we must add 0x10 to one of the tens nibble operators (line 33) and subtract ten from the result of the ADD instruction (line 34). At this point, the solution completes the ones nibble.

- The code starting at the label "lt_10x" essentially repeats the same action of the code starting at the "add_ones" label, but does it with the tens nibble. Because the code is similar, we'll save you the boredom of repeating the description. The section of code increments the 100's nibble if necessary also.

- This code is somewhat tricky in that when dealing with the high nibble, there is a chance you will generate a carry-out. In this case, you will need to increment the 100's digit, but also subtract a value of six from the result of the 10's addition. But if you don't generate a carry, you'll also may have a value greater 10 and less than 15. This case is similar to the 1's digit where we adjust it by simply incrementing the higher digit and subtracting 10 from the lower digit.

- The final section of the code clears the 100s nibble if necessary (line 48). This section of code also reformats the ones and ten's nibble into one register by first copying the previous ones nibble result and then ORing it with the (line 50) the tens nibble result.

Example 15-28:

Write a RAT assembly language subroutine that converts a 2-digit BCD number into an unsigned binary number. The 2-digit BCD number resides in r5 with the upper and lower nibbles representing the 10's and 1's digit, respectively. Place the result in r8. This subroutine should not change the value in r5.

Solution:

```
(00) ;-------------------------------------------------------------
(01) ;- subroutine: Bcd2bin
(02) ;-
(03) ;- This subroutine converts a two 2-digit BCD numbers residing in
(04) ;- r5 to an unsigned binary number. The upper and lower nibble
(05) ;- in r5 represent the 10's and 1's digit, respectively. The
(06) ;- resulting unsigned binary number is placed in r8; this
(07) ;- subroutine does not change the value of r5.
(08) ;-
(09) ;- Affected regs & flags: r0, r8, C, Z
(10) ;-------------------------------------------------------------
(11) .EQU LO_NIB_MASK    = 0x0F
(12) .EQU HI_NIB_MASK    = 0xF0
(13) .EQU LO_TEN         = 0x0A
(14) .EQU HI_TEN         = 0xA0
(15)
(16) Bcd2bin:
(17) init:       MOV     r0,r5            ; copy data
(18)             MOV     r8,r5            ; copy data
(19)             AND     r8,LO_NIB_MASK   ; lo nib is correct bin val
(20)
(21)             LSR     r0               ; move top nibble to low
(22)             LSR     r0               ;  nibble
(23)             LSR     r0
(24)             LSR     r0
(25)             AND     r0,LO_NIB_MASK   ; clear top nibble
(26)
(27) loop:       CMP     r0,0x00          ; keep adding 10 for each
(28)             BREQ    done             ;  value in 10's digit
(29)             ADD     r8,LO_TEN        ; accumulate 10
(30)             SUB     r0,0x01          ; decrement 10's count
(31)             BRN     loop             ; keep doing stuff
(32)
(33) done:       RET
(34) ; -------------------------------------------------------------
```

Figure 15-33: Solution for Example 15-28.

Notes Regarding the Solution: Fun stuff embedded in the solution. T

- Lines 17 & 18 of the solution copy the original BCD value into two working registers. The notion here is that we'll need to operate on both nibbles in the BCD number in order to complete the number conversion. Copying the number saves us the trouble of pushing and popping the original value on and off the stack.

- Line 19 clears the top nibble of register r8. Since the lower nibble can be no larger than nine, the value in the lower nibble represents the value that interests us. What we'll do later in the solution is to count the number of "tens" in the upper nibble and add one ten to r8 for each ten. In this way, we'll use register r8 as an accumulator.

- The code in lines 21-25 shifts the upper nibble to the lower nibble. Since we're using the LSR instruction, we don't know what's in the C flag. Because of this, we clear the top nibble after the four left-shifts are complete; this ensures we'll only be working with the original upper nibble (now in the lower nibble position).

- The code starting with the "loop" label is essential a multiply by ten operation. We essentially need to multiply the top nibble by ten and add it to the lower nibble. We're doing a multiply by doing consecutive additions: we add ten to the register we're using as an accumulator (r8) for each loop iteration. The iterative count for the loop is the value of the original tens value (the upper nibble) which is now in the lower nibble position.

- When the loop exhausts its iterative count, the value in register r8 is the converted value; the subroutine subsequently returns.

Example 15-29:

Consider the row of eight LEDs on a given development board. Write an RAT Assembly language program that will make it appear as if one lit LED is bouncing back and forth on the LEDs. One LED should always be on. For this program, you can call a subroutine named "Delay", which provides the required time delay so the LED is moving at a good speed; you need to call this subroutine but you don't need to write it. Your program should contain no more than 20 instructions. Consider the LED port to be port 0x20.

Solution:

```
(00) ;-------------------------------------------------------------------
(01) ;- Program: bounce
(02) ;-
(03) ;- This programs makes it appear as if one lit LED is bouncing
(04) ;- back and forth between the eight LEDs on the development
(05) ;- board. This program calls an imaginary subroutine that slows
(06) ;- the bouncing rate down to something humans can see and
(07) ;- comprehend, though the concept may still be too complicated
(08) ;- for academic administrators to comprehend.
(09) ;-
(10) ;-------------------------------------------------------------------
(11) .EQU LED_PORT = 0x20             ; port for LED output --- OUTOUT
(12) ;-------------------------------------------------------------------
(13) .CSEG
(14) .ORG  0x10
(15)
(16) init:    MOV   r8,0x01          ; init the one lit LED
(17)          CLC                    ; make sure the carry flag is '0'
(18)
(19) main:    MOV   r7,0x07          ; load loop count
(20) loop1:   OUT   r8,LED_PORT      ; output LED data
(21)          CALL  Delay            ; wait
(22)          LSL   r8               ; shift left
(23)          SUB   r7,0x01          ; decrement loop count
(24)          BRNE  loop1            ; continue if need be
(25)
(26)          MOV   r7,0x07          ; load loop count
(27) loop2:   OUT   r8,LED_PORT      ; output LED data
(28)          CALL  Delay            ; wait
(29)          LSR   r8               ; shift right
(30)          SUB   r7,0x01          ; decrement loop count
(31)          BRNE  loop2            ; loop if necessary
(32)
(33)          BRN   main             ; lather, rinse, repeat
(34) ;-------------------------------------------------------------
(35)
(36) ;-------------------------------------------------------------
(37) ;- Subroutine: Delay
(38) ;-
(39) ;- This is a stub; it does nothing useful, similar to an
(40) ;- academic administrator.
(41) ;-------------------------------------------------------------
(42) Delay:  AND   r0,r0
(43)         RET
(44) ; -------------------------------------------------------------
```

Figure 15-34: Solution for Example 15-29.

Notes Regarding the Solution: Fun stuff embedded in the solution. The basic form of this solution is that there are two iterative loops: one each for making the lit LED appears to move from left-to-right and from right-to-left. Wait for it in the solution that follows.

- The solution mentions/uses the notion of a "stub" for the Delay subroutine. As written, the subroutine does nothing useful; we place it there just in case we want to fill in an appropriate delay later. The notion of a "stub" comes up often in programming circles, particularly when you're using a top-down design approach.

- The initialization portion of the code (the line with "init" label) comprises of writing an initial value to a register that we later want to use to actuate the one lit LED. The solution does not actually turn on the LED until the "loop1" iterative loop. The way we do this here is arbitrary, meaning we could have done it differently. Additionally, we'll first turn on the right-most LED (associated with the one on bit in the value 0x01) and start shifting it left. This problem did not state this and is thus arbitrary.

- The code associated with the "loop1" iterative loop first turns on the LED by outputting the value in the working register r8 to the development board's LEDs (line 19). This loop first turns on the LED, delays, shifts the working value to the left, then checks the iterative count. This continues for seven iterations, thus making the one lit LED travel from the right-most to the left-most LED on the development board.

- The code associated with the "loop2" label is similar to the code associated with the "loop1" label so we won't describe it again here in painful detail. The difference between the code associated with "loop1" and "loop2" is that "loop1" code shifts the LED from right-to-left while the "loop2" code shifts the lit LED from left-to-right.

- When the iterative count in "loop2" runs out, (when it equals zero), the program branches back to the "main" label, which is the code, associated with the left shifting.

Example 15-30:

Write an interrupt driven RAT assembly language program that has both a foreground task and a background task. The foreground task constantly is reading from port 0x22, complementing the data, and writing the data to port 0x23. The background task implements a 2-interrupt delay. When the RAT MCU receives an interrupt, it reads a value from port 0x55. The value that was read from port 0x55 two interrupts earlier is then output to port 0x57. Keep your interrupt routine as short as possible. Your program should output 0xFF for the first two writes in the background task. Your program should contain no more than 15 instructions.

Solution:

```
(00) ;-----------------------------------------------------------------
(01) ;- Program: Interrupt Delay Thang
(02) ;-
(03) ;- This program constantly reads from port 0x22, complementing
(04) ;- the data, and writing the data to port 0x23. The background
(05) ;- task implements a 2-interrupt delay. When the RAT MCU receives
(06) ;- an interrupt, it reads a value from port 0x55. The value that
(07) ;- was read from port 0x55 two interrupts earlier is then output to
(08) ;- port 0x57. The program outputs 0xFF for the first two writes in
(09) ;- the background task.
(10) ;-----------------------------------------------------------------
(11) .CSEG
(12) .ORG  0x10
(13)
(14) init:    MOV   r20,0xFF         ; init ISR delay registers
(15)          MOV   r21,0xFF         ;
(16)          SEI                    ; allow interrupts
(17)
(18) main:    IN    r0,0x22          ; get some data
(19)          EXOR  r0,0xFF          ; complement data
(20)          OUT   r8,0x23          ; output LED data
(21)          BRN   main             ; repetition is a good thing
(22) ;-------------------------------------------------------------
(23)
(24) ;-------------------------------------------------------------
(25) ;- Subroutine: ISR
(26) ;-
(27) ;- This subroutine implements the interrupt service routine.
(28) ;- This ISR implements a two interrupt delay. During this ISR,
(29) ;- a value is read from an input port; the value read two
(30) ;- interrupts earlier is output. The first two output values
(31) ;- output are 0xFF.
(32) ;
(33) ;- Affected regs & flags: r20, r21, r22
(34) ;-------------------------------------------------------------
(35) ISR:
(36)         OUT    r20,0x57         ; output oldest data
(37)         MOV    r20,r21          ; transfer data to create delay
(38)         MOV    r21,r22
(39)         IN     r22,0x55         ; get new data
(40)         RETIE                   ; go back to foreground task
(41) ;-------------------------------------------------------------
(42)
(43) ;-------------------------------------------------------------
(44) .CSEG
(45) .ORG  0x3FF
(46)         BRN   ISR:
(47) ; -------------------------------------------------------------
```

Figure 15-35: Solution for Example 15-30.

Notes Regarding the Solution: This is somewhat of a stupid problem, but it does have one very important consideration. I've often asked students this type of question on exams: half the class did the problem perfectly, the other half of the class was clueless on what to do. The notion of creating a delay such as the one in this example only becomes easy after you figure it out or someone tells you what to do. This solution represents an act of someone telling you what to do; plan to put the solution in your bag of assembly language programming tricks. Here is some other fun stuff embedded in the solution.

- The initialization portion of the program (the three instructions starting at the "init" label) does something seemingly cryptic. There appears to be no reason to initialize the registers r20 & r21 to 0xFF, but there is a really good reason for doing this. The reason does not become apparent until we discuss the ISR, which we'll do in a later bullet.

- The main code is definitely pointless, as it implements a procedure that is similar to one of the first programming examples in this chapter. That being the case, we'll not discuss it here.

- Lines 45 & 46 are assembler directives that you must use to ensure an unconditional branch to the ISR lives at the vector address of the RAT MCU. The solution first makes sure it's in the code segment with the .CSEG directive. This is not necessary but it makes the code more robust; the solution is in the code segment already at this point in the code. The important and required directive is the .ORG directive; we must use this directive to make sure the assembler places the next instruction at address 0x3FF in program memory. This is of course the vector address for the RAT MCU's one interrupt.

- Once we've done all the stuff in the previous bullet, we then place the unconditional branch instruction at the vector address. In this way, when the RAT MCU receives an interrupt, program control transfers to the instruction at prog_rom address 0x3FF, and then transfers to the ISR by the unconditional branch instruction that the source code placed at address 0x3FF.

- The ISR is solely responsible for implementing the three-interrupt delay function. The idea here is to output the value input three interrupts earlier. The code does this by outputting the date received three interrupts earlier, and then shifting the values from pre-determined registers. The notion of a "bucket brigade" comes to mine (look it up on Wikipedia). The ISR essentially implements a three-position FIFO (first-in, first out) using registers r20, r21, and r22. Recall that a stack was a "LIFO" (last-in, first-out).

Example 15-31:

Write a RAT assembly language subroutine checks to see if a number is within a specified range. The specified range of the number is given in r10 and r11; no ordering on these numbers is stated. If the number in r13 is within the range (not inclusive) of the two stated numbers, then bit0 of r15 is set; otherwise, bit0 is cleared. Do not alter the contents of either r10 or r11. Provide flowchart for your solution.

Solution:

```
(00) ;-------------------------------------------------------------
(01) ;- subroutine: Check_range
(02) ;-
(03) ;- This subroutine checks to see if a number is within the
(04) ;- range specified in r10 and r11. This subroutine first
(05) ;- sorts the values in r10 and r11 if necessary. If the value
(06) ;- in r13 is within the range (not inclusive) of the two
(07) ;- values, then bit0 of r15 is set; otherwise, bit0 is cleared.
(08) ;-
(09) ;- Affected regs & flags: r12, r15, C, Z
(10) ;-------------------------------------------------------------
(11) .EQU B0_CLR_MASK    = 0xFE
(12) .EQU B0_SET_MASK    = 0x01
(13)
(14) Check_range:
(15) init:        PUSH    r10             ; save values
(16)              PUSH    r11
(17)
(18) sort:        CMP     r11,r10         ; see if r11 < r10
(19)              BRCC    no_sort         ; don't swap
(20)              MOV     r12,r10         ; reg swap code
(21)              MOV     r10,r11
(22)              MOV     r11,r12
(23)
(24) no_sort:     CMP     r13,r11         ; is r13 < r11?
(25)              BRCC    clr_bit
(26)
(27)              CMP     r10,r13         ; is r13 > r10
(28)              BRCC    clr_bit
(29)
(30) set_bit:     OR      r15,B0_SET_MASK ; set bit
(31)              BRN     done
(32)
(33) clr_bit:     AND     r15,B0_CLR_MASK ; clear bit
(34)
(35) done:        POP     r11             ; restore values
(36)              POP     r10
(37)              RET                     ; go west young person
(38) ; -------------------------------------------------------------
```

Figure 15-36: Solution for Example 15-31.

Notes Regarding the Solution: This is somewhat of an interesting problem in that it combines both a simple sort and a couple of comparisons. Here is some other fun stuff embedded in the solution.

- Here is the overall approach this solution takes: save values, sort range, check value against range, tweak bits, restore values, and return. Breaking problems down like this can often help; I hope this helps.

- The problem states not to change the values in register r10 & r11, so the solution pushes these values onto the stack at the beginning of the subroutine and restored before the subroutine returns control back to the calling program.

- The code on lines 18-22 (starting at the “sort” label) performs a sort of the two range values, as the problem states the two values are not in any particular order. The CMP operation assumes r11 is greater than r10, but if this is not so, the C flag is set, which makes the conditional branch fail, and the code then swaps register r10 & r11.

- Lines 24-28 compare the value against the two previously sorted values to establish if the value is within the range of the sorted values. There are two checks: the value must be less than the upper value and greater than the lower value. If the values meet these two conditions, the LSB in r15 is set; otherwise, the code clears the LSB in r15.

- You're in luck: Figure 15-37 provides a viable flowchart that models the code in this solution. Keep in mind that this is only one of many possible flowcharts that would adequately model the code in this solution.

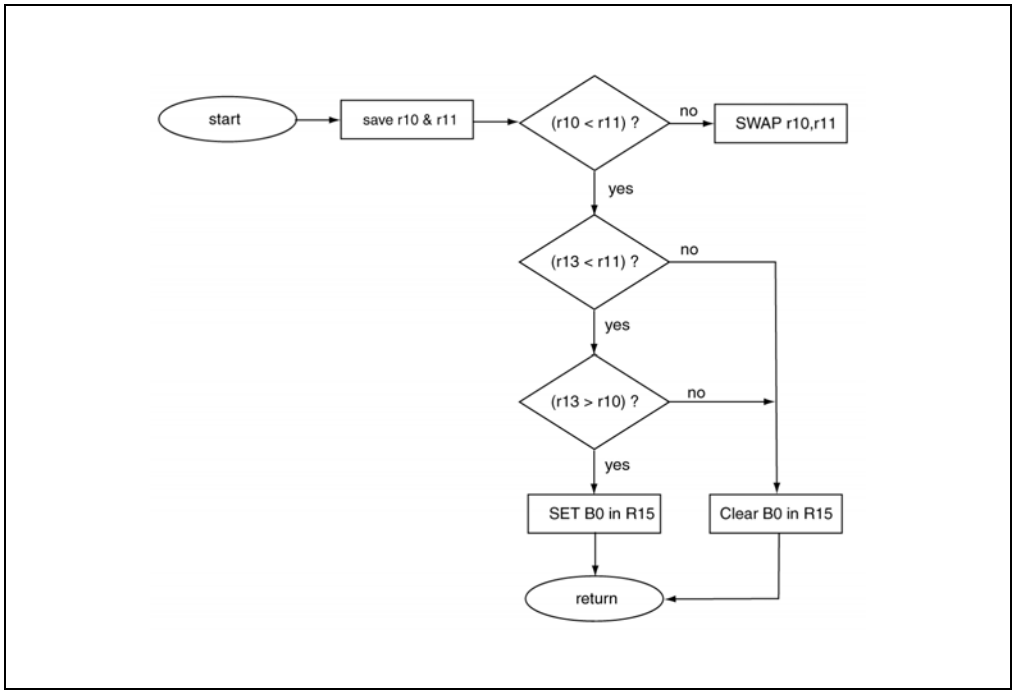


Figure 15-37: A flowchart for the solution of Example 15-31.

Example 15-32:

Write a RAT interrupt driven program that implements a simple digital filter. The main code counts the number of interrupts and outputs this binary value to port 0x88 each time the RAT receives an interrupt. This count is an 8-bit binary and should rollover to zero when the count reaches 0xCC. When the RAT receives an interrupt, it reads a value from port 0x98. This value is averaged with the value read from this input port during the previous interrupt and the averaged result is written to port 0x99. The program rounds up the averaging calculation. Keep you interrupt service routine as short as possible.

Solution:

```
(00) ;-----------------------------------------------------------------
(01) ;- Program: Simple Averaging Filter
(02) ;-
(03) ;- This RAT interrupt driven program implements a digital filter.
(04) ;- The main code counts the number of interrupts received and
(05) ;- outputs this binary value to port 0x88 each time the RAT
(06) ;- receives an interrupt. This count is an 8-bit binary and rolls
(07) ;- over to zero when the count exceeds 0xCC. When the RAT receives
(08) ;- an interrupt, it reads a value from port 0x98; this value is
(09) ;- averaged with the value read during the previous interrupt
(10) ;- and the result is written to port 0x99.
(11) ;-----------------------------------------------------------------
(12) .CSEG
(13) .ORG  0x10
(14)
(15) init:    MOV   r20,0x00         ; init ISR register
(16)          MOV   r16,0x00         ; clear flag ISR flag register
(17)          MOV   r10,0x00         ; clear ISR counter register
(18)
(19)          OUT   r10,0x88         ; output ISR count value
(20)          SEI                    ; allow interrupts
(21)
(22) main:    TEST  r16,0x01         ; see if interrupt flag set
(23)          BREQ  main
(24)
(25) incr:    ADD   r10,0x01         ; increment ISR count value
(26)          CMP   r10,0xCC         ; max count exceeded?
(27)          BRCC  no_rst
(28)          MOV   r10,0x00         ; reset ISR count
(29)
(30) no_rst:  OUT   r10,0x88
(31)          SEI
(32)          BRN   main             ; repetition is real good
(33) ;-----------------------------------------------------------
(34)
(35) ;-----------------------------------------------------------
(36) ;- Subroutine: ISR
(37) ;-
(38) ;- This subroutine implements the interrupt service routine.
(39) ;- This ISR implements an averaging filter which averages
(40) ;- the two more recent values read from during the ISRs. The
(41) ;- calculation includes a rounding up feature.
(42) ;-
(43) ;- Affected regs & flags: r20, r21, r16, r30
(44) ;-----------------------------------------------------------
(45) ISR:
(46)         MOV    r20,r21          ; save more recent data
(47)         IN     r21,0x98         ; get new data
(48)         MOV    r30,r20          ; copy old data
(49)         ADD    r30,r21          ; add new data
(50)         CLC                     ; put C flag in known state
(51)         LSR    r30              ; divide by 2
(52)         ADDC   r30,0x00         ; add the carry for roundup
(53)         OUT    r30,0x99         ; output the average
(54)
(55)         MOV    r16,0x01         ; set an interrupt flag
(56)         RETID                   ; go back to foreground task
(57) ;-----------------------------------------------------------
(58) ;-----------------------------------------------------------
(59) .CSEG
(60) .ORG  0x3FF
(61)         BRN   ISR
(62) ; -----------------------------------------------------------
```

Figure 15-38: Solution for Example 15-32.

Notes Regarding the Solution: Once again, there are many different interesting aspects to this problem. The problem is essentially a digital filter with features, where some features are more interesting than others are. Here's a discussion of some of the more fun stuff embedded in the solution.

- In problems such as these, you must have faith that there is some device out there connected to the RAT MCU that officially generates interrupts. We omit these details, as they are not pertinent to the programming aspect of the problem.

- The solution starts with initialization code; the code clears the interrupt flag and count registers (lines 15-17). The code also outputs initial count of interrupts. Finally, the code enables the interrupts, which allows the code to actually work as desired.

- The foreground task (the non-ISR code) comprises of two parts. The first part is an infinite loop (lines 22-23) that checks the designated "flag" that indicates an interrupt was processed. When the code detects interrupt processing, it drops down to the "deal with" interrupt counter portion of the code. The "deal with" section includes incrementing a counter and verifying that the counter has not exceeded the maximum value stated in the problem description. Nothing too exciting here.

- The ISR is responsible for implementing the digital filter. This filter essentially stores the value input from the previous interrupt and averages it with the value input in the current interrupt. The solution does the averaging using the LSR instruction, which essentially performs the required divide by two. Because the problem statement requests a round-up, the ISR also adds one to the divide-by-two result if the bit that shifted out the right end of the register (the LSB before the shift) was a '1'.

- Before exiting the ISR, the code sets the LSB in register 16 (line 55), which acts to notify the foreground task that an interrupt has occurred. The ISR code returns with the interrupt disabled, which is done because we need to do further processing in the foreground code before we re-enable the interrupts.

Example 15-33:

Write a RAT assembly language subroutine that converts an 8-bit unsigned binary number into a 3-digit BCD value. The unsigned binary number is provided in register r20. Place the resultant BCD number in r25 & r26, where the lower-nibble of r26 holds the 100's digit, the upper-nibble of r25 holds the 10's digit, and the lower nibble of r25 holds the 1's digit. This subroutine should not alter the value in r20.

Solution:

```
(00)  .CSEG
(01)  .ORG  0x10
(02)
(03)  main:    MOV   r20,0x64    ; test driver
(04)           CALL  Bin2bcd
(05)           BRN   main
(06)  ;-----------------------------------------------------------------
(07)  ;- subroutine: Bin2bcd
(08)  ;-
(09)  ;- This subroutine converts an 8-bit unsigned binary number
(10)  ;- into a 3-digit BCD value. The unsigned binary number is
(11)  ;- provided in r20 and the resultant BCD number in r25 & r26,
(12)  ;- where the lower-nibble of r26 holds the 100's digit, the
(13)  ;- upper-nibble of r25 holds the 10's digit, and the lower
(14)  ;- nibble of r25 holds the 1's digit. This subroutine does
(15)  ;- not alter the value in r20.
(16)  ;-
(17)  ;- Affected regs & flags: r25, r26, C, Z
(18)  ;-----------------------------------------------------------------
(19)  .EQU CIEN     = 0x64
(20)  .EQU TEN      = 0x0A
(21)
(22)  Bin2bcd:
(23)  init:       PUSH    r20             ; save the value
(24)              MOV     r25,0x00        ; clear result register
(25)              MOV     r26,0x00
(26)
(27)  hunds:      CMP     r20,CIEN        ; comare with 100
(28)              BRCS    tens            ; go onwards if no 100's
(29)              ADD     r26,0x01        ; increment 100's count
(30)              SUB     r20,CIEN        ; adjust original value
(31)              BRN     hunds           ; keep doing it
(32)
(33)  tens:       CMP     r20,TEN         ; comare with 10
(34)              BRCS    ones            ; go onwards if no 10's
(35)              ADD     r25,0x10        ; increment 10's count
(36)              SUB     r20,TEN         ; adjust original value
(37)              BRN     tens            ; keep doing it
(38)
(39)  ones:       OR      r25,r20         ; add the 1's sloppy seconds
(40)
(41)  done:       POP     r20             ; restore the original data
(42)              RET                     ; go back, all the way back
(43)  ; ---------------------------------------------------------------
```

Figure 15-39: Solution for Example 15-33.

Notes Regarding the Solution: This is another classic problem that involves converting a number from one format to another. Problems such as these are nice for two reasons. First, they are actually quite useful out there in computer land as different devices have their own way of representing numbers. For example, the BCD number in this problem may be used for some type of display such as a 7-segment display, while the MCU stores the number itself in one (or more) of its internal registers. The second reason this problem is nice is that it's a real problem in that you can relate to it. You know decimal notions (such as BCD) and you know various binary representations, so it rather intuitive to switch between the formats. Here is some other fun stuff embedded in the solution.

- The solution include a calling program (the first six lines of the solution) so you can step it through a simulator if you are so inclined. Have lots of fun.

- The problem states that the subroutine must protect the value in r20, so the solution pushes it onto the stack at the start of the subroutine and pops it off the stack before the subroutine exits. This is part of the subroutine initialization. Other initialization includes the clearing of the two result registers r25 & r26 (lines 24-25).

- There are many ways you can solve this problem. The algorithm this solution uses starts by subtracting as many “100s” as possible, then as many “10s” as possible; what remains at this point is the ones.

- The code starting at the label “hunds” subtracts as many 100s from the value as possible. Each time it successfully subtracts a value, it increments the 100s nibble in the designated register (register r26 on line 29).

- The code then performs the same basic operation but this time using a 10s subtraction, as the resulting value from the previous operation is guaranteed to be less than 100. The solution subtracts as many 10s as possible from the value and increment Note that the solution does not actually increment the value in the register; it adds 0x10, which is essentially incrementing the upper nibble (the 10’s nibble) of the converted value. Doing it this way saves us the trouble of shifting the 10s count from the lower nibble to the upper nibble.

- Whatever is left over in register r20 is ORed with the 10s nibble (the upper nibble of r25). We don’t know or care what is in r20 now (it may be anything from zero to nine); we simply use the OR instruction to include the result with the upper nibble. At this point, we restore the original value in register r20 and exit the subroutine.

Example 15-34:

Write a RAT assembly language subroutine that takes the absolute value of a signed binary number. The value in r10 & r11 represents a 16-bit unsigned binary number in 2’s complement form, with r11 being the upper eight bits and r10 the lower eight bits. This subroutine returns with the magnitude of the 16-bit number in r10 & r11. Also provide a flowchart for the solution.

Solution:

```
(00) ;-------------------------------------------------------------------
(01) ;- subroutine: Mag16bit
(02) ;-
(03) ;- This subroutine that takes the absolute value of a signed binary
(04) ;- number. The value in r10 & r11 holds a 16-bit unsigned binary
(05) ;- number in 2’s complement form, with r11 being the upper eight
(06) ;- bits and r10 the lower eight bits. This subroutine returns with
(07) ;- the magnitude of the 16-bit number in r10 & r11.
(08) ;-
(09) ;- Affected regs & flags: r10, r11, C, Z
(10) ;-------------------------------------------------------------------
(11) .EQU SIGN_MASK  = 0x80
(12)
(13) Mag16bit:   TEST    r11,SIGN_MASK    ; see if value is negative
(14)             BRNE    two_comp         ; keep going if negative
(15)             RET                      ; return if positive
(16)
(17) two_comp:   EXOR    r11,0xFF         ; complement upper byte
(18)             EXOR    r10,0xFF         ; complement lower byte
(19)             ADD     r10,0x01         ; add one to low byte
(20)             ADDC    r11,0x00         ; propagate one to upper byte
(21)             RET                      ; go back back back back back
(22) ; ------------------------------------------------------------------
```

Figure 15-40: Solution for Example 15-34.

Notes Regarding the Solution: This is another example problem that requires you to know both programming concepts as well as basic digital design concepts. There are actually two issues with this problem. First, the RAT MCU does not have any special way of handling signed numbers of any time. Second, the RAT MCU does not have any direct way of handling 16-bit numbers. We must consider these two items when we formulate the solution. Here is some other stuff of interest regarding this problem.

- This is an absolute value problem, so the first thing we must determine is whether the given 16-bit number is negative or positive. We do this by check the MSB of the upper eight bits, which the problem stores in register r11. The first two lines of the solution (lines 13-14) perform this check. We only need to do something if the resulting number is negative, so the solution returns if the number is positive. Recall that the MSB of the more significant byte is the sign-bit for the 16-bit value (in other words, for both r10 & r11).

- When the 16-bit number is negative, we must perform a 2's complement on the entire 16-bits. It turns out that this is no more difficult than performing the 2's complement on an 8-bit value. We use the notion that a 2's complement is the 1's complement (complementing all the bits) and adding one. Lines 17 & 18 of the solution perform the complement using the XOR instruction on both the upper and lower eight bits of the 16-bit value. We then must add one to the least significant byte. Adding one to this byte may result in a carry, which we must take into account when we add one to the most significant byte. To handle the carry, we use an ADDC instruction when we increment the upper byte.

- The approach this solution takes is overall straightforward. As proof of this, you can examine the flowchart in Figure 15-41. Note in this flowchart there is only one entry point (the terminal symbol with the "start") but there are two exit points (the two terminal symbols with the "return"). This is a fine flowchart indeed, in that it has an appropriate level of detail. The fact that it does not mention any RAT MCU instructions is not a problem.

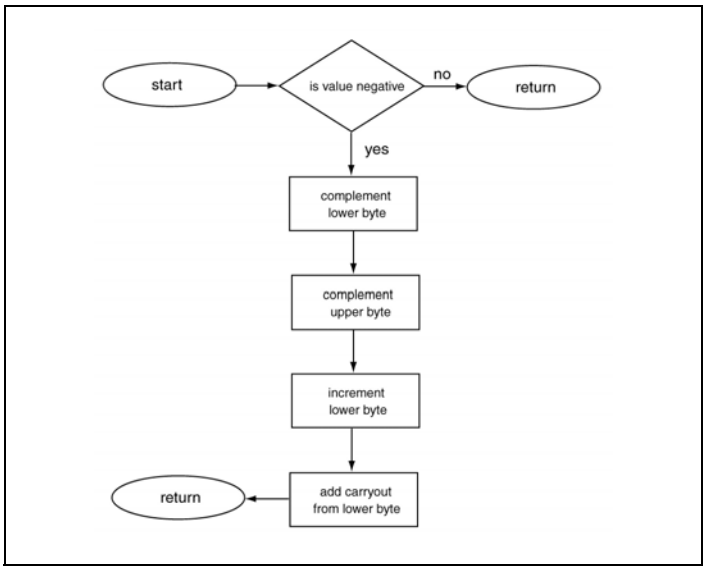


Figure 15-41: A flowchart modeling the given solution to Example 15-34.

Example 15-35:

Write a RAT assembly language subroutine that clamps and 8-bit unsigned binary number into the range [33,233]. This means if the number is in the given range, it is not altered. If the number is less than the lower bound, the number is clamped to the lower bound. If the number is greater than the upper bound, the number is clamped to the upper bound. The binary value is provided in r22; the resultant value remains in this register.

Solution:

```
(00)  ;-------------------------------------------------------------------
(01)  ;- subroutine: Clamp_val
(02)  ;-
(03)  ;- This subroutine clamps the 8-bit unsigned binary number in r22
(04)  ;- into the range [33,233]. If the number is in the given range,
(05)  ;- it is not altered. If the number is less than the lower bound,
(06)  ;- the number is clamped to the lower bound. If the number is
(07)  ;- greater than the upper bound, the number is clamped to the upper
(08)  ;- bound. The resultant value remains in r22.
(09)  ;-
(10)  ;- Affected regs & flags: r22, C, Z
(11)  ;-------------------------------------------------------------------
(12)  .EQU LO_VAL  = 0x21
(13)  .EQU HI_VAL  = 0xE9
(14)
(15)  Clamp_val:  CMP     r22,LO_VAL       ; check lower bound
(16)              BRCC    not_low          ; not too low, check hi-ness
(17)              MOV     r22,LO_VAL       ; replace value with lo clamp
(18)              RET                      ; go back now
(19)
(20)  not_low:    CMP     r22,HI_VAL       ; check upper bound
(21)              BRCS    not_hi           ; not above clamp
(22)              MOV     r22,HI_VAL       ; replace value with hi clamp
(23)
(24)  not_hi:     RET                      ; go go go back back back
(25)  ; -----------------------------------------------------------------
```

Figure 15-42: Solution for Example 15-35.

Notes Regarding the Solution: This is a practical little problem; often times in programming land you need to adjust values to fit within a given range. The basic form of the solution is to perform two checks: one for the upper and lower bound. If either of these two checks fails, the solution adjusts the sent value and then exits the subroutine. If the value does not require adjusting, subroutine exits without making any changes. Here is some other fun stuff embedded in the solution.

- The solution uses assembler directives to handle both the upper and lower clamping values. This represents good programming as some external entity may require that these values change. Having these values in one spot, such as at the beginning of the program, makes changes such as these easier, faster, and better.

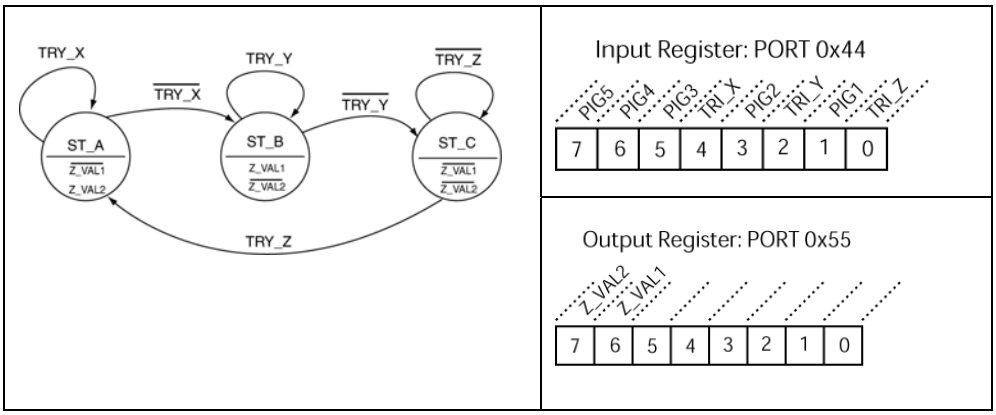
- The first part of the solution checks if the sent value is less than the lower bound by using a CMP instruction (line 15). If the sent value is in fact less than the lower threshold, it is set equal to the clamped value (line 17) before exiting the subroutine.

- The second part of the solution checks if the sent value is greater than the upper bound, also by using a CMP instruction (line 20). If the sent value is greater than the upper threshold, the sent value is set equal to the upper clamp value before exiting the subroutine.

- If the sent value is within the given range, both the upper and lower clamp checks fail and the subroutine exits. Note that the solution shares the final RET instruction (line 24) with the failing of the upper bound check.

Example 15-36:

Write a RAT assembly language program the implements the following FSM.



Solution:

```
(00) ;-------------------------------------------------------------------
(01) ;- This program is the solution to a RAT practice programming.
(02) ;- The code below implements a firmware version of an FSM.
(03) ;-------------------------------------------------------------------
(04) .EQU TRY_X_MASK    = 0x10
(05) .EQU TRY_Y_MASK    = 0x04
(06) .EQU TRY_Z_MASK    = 0x01
(07)
(08) .CSEG
(09) .ORG  0x10
(10)
(11) main:
(12) ST_A:       MOV     r1,0x80          ; load out data for ST_A
(13)             OUT     r1,0x55          ; output Moore data
(14) test1:      IN      r0,0x44          ; get input data
(15)             TEST    r0,TRY_X_MASK    ; check bit(4)
(16)             BRNE    test1
(17)
(18) ST_B:       MOV     r1,0x40          ; load out data for ST_B
(19)             OUT     r1,0x55          ; output Moore data
(20) test2:      IN      r0,0x44          ; get input data
(21)             TEST    r0,TRY_Y_MASK    ; check bit(2)
(22)             BRNE    test2
(23)
(24) ST_C:       MOV     r1,0x00          ; load out data for ST_C
(25)             OUT     r1,0x55          ; output Moore data
(26) test3:      IN      r0,0x44          ; get input data
(27)             TEST    r0,TRY_Z_MASK    ; check bit(0)
(28)             BREQ    test3
(29)
(30)             BRN     ST_A             ; transition to ST_A
(31) ; -------------------------------------------------------------
```

Figure 15-43: Solution for Example 15-36.

Notes Regarding the Solution: It's inevitable that if you program in assembly language and you're actually doing meaningful stuff, you'll have to model a FSM in assembly language. This problem presents such a state machine. Therefore, if you don't understand finite state machines, you'll probably have a tough time understanding this problem. If that is the case, go back and review how state machines work, as this problems assumes that you already have such knowledge. Outside of these rude comments, here is some fun stuff embedded in this problem's solution.

- Though you may not realize it, this is actually close to being a good example. Out there in MCU-land, people use single bits to represent various inputs and outputs. This problem only uses the "TRI_?" inputs; the other inputs are there for appearance purposes only. Most important thing in this problem is to realize the inputs govern the state transitions and that the outputs are all Moore outputs.

- This problem also uses a fair amount of bit masking. This is good programming practice because if the hardware changed (meaning the status bits changed position), you'd have to spend a bunch of time making sure you found every occurrence of the change throughout your code. You don't want to do that; no one wants to waste such time.

- Note that the structure of the code is that there is one section of code per state in the given FSM. This is typical for this type of problem. Embedded in each of these sections of code are conditional loops, which check the status inputs and outputs the control outputs. More importantly is that the code only outputs the Moore values once per state (or in this case, once per section of code).

- The first section of code monitors the "TRY_X" input; as long as this input is a '1', the code stays in this state. Once this input is a '0', the FSM needs to change state, which means the check in this section of code fails and moves onto the next section of code. Before the code enters the input check loop, it first outputs the value of the Moore outputs. In the first section of code, this includes writing the 0x80 to the output port. The value of 0x80 provides a Z_VAL1 = '0' and Z_VAL2 = '1'. The other two sections of the code are similar to the first section so we'll not bore you with the gory details.

Example 15-37:

Write a RAT assembly language subroutine that arranges the values in the four contiguous memory locations starting at 0x40 into ascending order (the lowest value should be at 0x40). Store the results back into the same four memory locations. For the record, we refer to this type of problem as sorting.

Solution:

```
(00) ;-----------------------------------------------------------------
(01) ;-  Subroutine: Sort_4val
(02) ;-
(03) ;- These subroutines sorts the four contiguous numbers in memory
(04) ; - starting at locations 0x40. The sort stores these numbers back
(05) ;- into the same registers.
(06) ;-
(07) ;- Affected regs & flags: possibly r0, r10, r11, r12, r13
(08) ;-----------------------------------------------------------------
(09) Sort_4val:
(10) init:      LD    r10,0x40         ; get values from memory
(11)            LD    r11,0x41
(12)            LD    r12,0x42
(13)            LD    r13,0x43
(14)
(15)            MOV   r0,0x03          ; do it one less than values to sort
(16)
(17) loop:
(18) cmp1:      MOV   r8,r13           ; sort the two higher order registers
(19)            MOV   r7,r12
(20)            CALL  sort_reg
(21)            MOV   r12,r7
(22)            MOV   r13,r8
(23)
(24) cmp2:      MOV   r8,r12           ; sort the next two registers
(25)            MOV   r7,r11
(26)            CALL  sort_reg
(27)            MOV   r11,r7
(28)            MOV   r12,r8
(29)
(30) cmp3:      MOV   r8,r11           ; sort the final two registers
(31)            MOV   r7,r10
(32)            CALL  sort_reg
(33)            MOV   r10,r7
(34)            MOV   r11,r8
(35)
(36)            SUB   r0,0x01          ; check iteration count
(37)            BRNE  loop
(38)
(39)            ST    r10,0x40         ; store values from memory
(40)            ST    r11,0x41
(41)            ST    r12,0x42
(42)            ST    r13,0x43
(43)
(44)            RET                    ; come on back to the farm
(45) ;-----------------------------------------------------------------
(46)
(47) ;-----------------------------------------------------------------
(48) ;-  Subroutine: sort_reg
(49) ;-
(50) ;- This subroutine arranges the values in r7 and r8 into ascending
(51) ;- order, meaning the value in r7 will be <= to value in r8.
(52) ;-
(53) ;- Affected regs & flags: possibly r7, r8, r9, C, Z
(54) ;-----------------------------------------------------------------
(55) sort_reg:  CMP   r8,r7      ; set flags with subtract
(56)            BRCC  done       ; if C=0, r8 >= r7; don't swap
(57)
(58)            MOV   r9,r8      ; swap registers
(59)            MOV   r8,r7
(60)            MOV   r7,r9
(61) done:      RET
(62) ;-----------------------------------------------------------------
```

Figure 15-44: Solution for Example 15-37

Notes Regarding the Solution: You've seen problems like this before, but in a slightly different format. Here are the advertised fun things embedded in the solution.

- Whenever you faced with any problem, you should keep two things in mind when you're formulating a solution. First, you want to think about the problem long enough to consider more than one path to the solution. The idea here is that you want to take the most efficient path. As you think about the problem, you'll gain more insight into the problem and most likely think of new and better ideas to solve the problem. If you immediately start coding the first solution you think of, it may lead to a programming dead end and you'll find yourself down an ugly path. Secondly, you want to make the solution as generic as possible. This means that if your know-nothing boss unexpectedly asks you to modify the problem and/or solution, you won't have to spend too much time pleasing the boss.

- The approach the solution takes is a simple sort; to be exact; the solution uses a "bubble sort". There are many different sorting algorithms out there; some are more "efficient" than the bubble sort. The good news about the bubble sort is that it is probably the most intuitive sorting algorithm out there. For this problem, we need to sort four items. In the worst-case scenario, the item on the bottom of the list needs to travel to the top of the list. Since there are four items that we need to sort, it would take three "iterations" to move something from the bottom of the list to the top of the list. The outline of this solution is thus "make three comparisons, and do it three times". An iterative construct holds the "three times" part of the algorithm; the body of the code in the iterative construct handles the "make three comparisons" portion. The solution structures the code such that you can easily see the parts of the algorithm we're discussing in this paragraph.

- To make the solution modular, we first want to design some sort of "sort" subroutine. The "sort_reg" subroutine handles the sorting of the registers. We previously discussed an algorithm such as this, so we'll spare you the details this time.

- Each portion of the three comparison sections makes a call to the "sort_reg" subroutine. Thus each section is responsible for loading the correct values into the working registers (r7 & r8) before the subroutine call and then loading them back into the other working registers (r10, r11, r12, r13) after the subroutine call.

Example 15-38:

Write a RAT assembly language program that attempts to do something meaningful as follows (use your imagination for the "meaningful" part). Consider that your program will be run on the development board with switches connected; assume the switches have been magically debounced by the debounce fairy (no relation to the tooth fairy but maintains email contact with Santa Claus). The right-most switch on the development board is used to enable interrupts, so if the switch is off, the interrupts are disabled. When the switch turns on, the first two interrupts are ignored; after that point, the ISR calls a special subroutine: do_something before returning to the foreground code.

Solution:

```
(00)  ;-----------------------------------------------------------------
(01)  ;- Problem: Write a RAT assembly language program that attempts
(02)  ;- to do something meaningful as follows (use your imagination
(03)  ;- for the “meaningful” part). Consider that your program will
(04)  ;- be run on the development board with switches connected; assume
(05)  ;- the switches have been magically debounced by the debounce
(06)  ;- fairy (no relation to the tooth fairy but maintains email
(07)  ;- contact with Santa Claus). The right-most switch on the
(08)  ;- development board is used to enable interrupts, so if the switch
(09)  ;- is off, the interrupts are disabled. When the switch turns on,
(10)  ;- the first two interrupts are ignored; after that point, the
(11)  ;- ISR calls a special subroutine: do_something before returning
(12)  ;- to the foreground task.
(13)  ;-----------------------------------------------------------------
(14)  .EQU SWITCHES  = 0x82
(15)
(16)  ;-----------------------------------------------------------------
(17)  init:    MOV    r30, 0x00       ; interrupt on flag variable
(18)           MOV    r31, 0x00       ; interrupt count variable
(19)           CLI                    ; ensure interrupts are dead
(20)
(21)  main:    IN     r0, SWITCHES    ; Input switch status
(22)           TEST   r0, 0x01        ; see if sw0 is on
(23)           BRNE   sw0_on          ; branch if switch is off
(24)           CLI
(25)           MOV    r30,0x00        ; clear interrupt on flag
(26)           MOV    r31,0x00        ; clear interrupt count
(27)           BRN    main            ; boo!
(28)
(29)  sw0_on:  SEI                    ; turn on interrupts
(30)           BRN    main            ; turn me on dead man
(31)  ;-----------------------------------------------------------------
(32)
(33)  ;-----------------------------------------------------------------
(34)  ;- Subroutine: do_something
(35)  ;- Description: Unfortunately, do_something does nothing
(36)  ;-
(37)  ;- Registers Tweaked: none
(38)  ;-----------------------------------------------------------------
(39)  do_something:
(40)           AND    r7, r7           ; has subliminal meaning (nop)
(41)           RET
(42)  ;-----------------------------------------------------------------
(43)
(44)  ;---- Interrupt Subroutine ---------------------------------------
(45)  ISR:     TEST    r30,0x01         ; check interrupt flag variable
(46)           BREQ    do_call          ; branch if flag set
(47)
(48)           ADD     r31,0x01         ; increment interrupt count
(49)           CMP     r31,0x03         ; seen two interrupts?
(50)           BREQ    turn_on          ; two seen; turn on interrupts
(51)           RETIE
(52)
(53)  turn_on: OR      r30,0x01         ; turn on interrupt flag
(54)
(55)  do_call: CALL    do_something
(56)           RETIE
(57)  ;-----------------------------------------------------------------
(58)
(59)  ;-----------------------------------------------------------------
(60)  ;- interrupt vector address
(61)  ;-----------------------------------------------------------------
(62)  .ORG 0x3FF
(63)           BRN   ISR               ; Go to ISR
(64)  ;-----------------------------------------------------------------
```

Figure 15-45: Solution for Example 15-38

Notes Regarding the Solution: What makes this an interesting problem is that there is a bunch of interesting communication between the foreground and background code. Communication of this type is quite common in assembly language programming-land, so you may as well get used to it. Besides this interesting communication, this problem is trying really hard to be a useful problem. Here is the other fun stuff embedded in the solution.

- This solution uses a register as a flag and another register as a counter. One flag indicates that the ISR is calling the special "do_something" subroutine (r30). The solution uses register r31 as a counter to count the number of interrupts that the program receives after the program enables the interrupts with the external switch. Since we are using these two registers, we must initialize them, which we do at the start of the program, which we indicate with the "init" label. Note that in the initialization sequence, the interrupts are disabled. Recall that the interrupts are masked or unmasked based on an external switch.

- The main code's primary function is to poll the switch; when the switch is on, the interrupts are unmasked. Yes, polling is bad, but this is a simple program that is fighting hard to do something meaningful. Please play along. The structure of the main code rather gives me the willies, in that I feel I could write it a bit more cleanly. The issue here is that it repeats instructions that it does not have to repeat (repeated instructions are always a sign that you could write the code more efficiently). I left the code as it was because there is really nothing useful the foreground task can be doing in this problem anyway.

- The ISR basic functionality is to check the interrupt count. Recall that we want to ignore the first two interrupts, which means we must use a register count how many valid interrupts have been received. Once the solution receives two interrupts, which of course means that the interrupts are enabled, the solution starts calling the "do_something" subroutine.

- The ISR always returns with the interrupts enabled. In this problem, it's the external switch that handles the masking and unmasking of interrupts.

- Because this is an interrupt driven program, we must place an unconditional branch instruction at the interrupt vector location. The code starting at line 59 does this. Note that there is not code after the code the code that sets the interrupt vector; if there was more code, we would have to issue another assembler directive so that any more code would appear in valid program code space.

Example 15-39:

Write a RAT assembly subroutine that searches a section of the scratch RAM to find the largest value in the particular section. The search start and end addresses are stored in register r25 and r26, respectively. Place the largest value in that range r30 and the address of that value in r31. Don't assume anything regarding the addresses in r25 & r26 (meaning, you had better check them before you do your algorithm). Make sure your subroutine does not alter values in r25 & r26.

Solution:

```
(00)  ;-----------------------------------------------------------------
(01)  ;- Problem: Write a RAT assembly subroutine that searches a
(02)  ;- section of the scratch RAM to find the largest value in the
(03)  ;- particular section. The search start and end addresses are
(04)  ;- stored in register r25 and r26, respectively. Place the largest
(05)  ;- value in that range r30 and the address of that value in r31.
(06)  ;- Don't assume anything regarding the addresses in r25 & r26
(07)  ;- (meaning, you better check them before you do your algorithm).
(08)  ;- Make sure your subroutine does not alter values in r25 & r26.
(09)  ;-----------------------------------------------------------------
(10)  .CSEG
(11)  .ORG 0x10
(12)  ;-----------------------------------------------------------------
(13)  init:    MOV    r25, 0x20       ; mem start address
(14)           MOV    r26, 0x10       ; mem end address
(15)
(16)
(17)  main:    CALL   Find_big        ; call the main thang
(18)           BRN    main            ; number 9, number 9...
(19)  ;-----------------------------------------------------------------
(20)
(21)  ;-----------------------------------------------------------------
(22)  ;- Subroutine: Find_big
(23)  ;-
(24)  ;- This subroutine finds the biggest values in scratch RAM in a
(25)  ;- range specified by [(r25),(r26)]. The subroutines stores the
(26)  ;- largest value in r30 and the address of that value in r31. This
(27)  ;- subroutine include addresses error checking for the wanker
(28)  ;- weedout affect.
(29)  ;-
(30)  ;- Affected regs & flags: r20, r30, r31, C, Z
(31)  ;-----------------------------------------------------------------
(32)  Find_big: CMP    r25,r26          ; return if they are not good.
(33)            BREQ   end              ; starting address must be at least
(34)            BRCC   end              ; one less than end address
(35)
(36)            PUSH   r25              ; save starting values
(37)
(38)  init:     LD     r30, (r25)       ; get the starting value
(39)            MOV    r31, r25         ; get the starting address
(40)
(41)
(42)  cont:     ADD    r25, 0x01        ; increment address
(43)            LD     r20, (r25)       ; get a new value
(44)
(45)            CMP    r30,r20          ; compare biggest and next value
(46)            BRCC   check            ;
(47)
(48)            MOV    r30,r20          ; store new big value
(49)            MOV    r31,r25          ; store new big address
(50)
(51)  check:    CMP    r25,r26          ; see if addresses are equal
(52)            BRNE   cont             ; end if they are equal
(53)
(54)            POP    r25              ; restore r25
(55)  end:      RET                     ; done at last...
(56)  ;-----------------------------------------------------------------
(57)
```

Figure 15-46: Solution for Example 15-39.

Notes Regarding the Solution: This problem uses many useful assembly language programming techniques. This is a classic search algorithm, where we search a list of things looking for the thing that has the attribute we're interested in. This problem presents rather generic information in the form of the addresses in memory you need to search. As a result, you can imagine that the solution must also be generic also. Finally, here is some other fun stuff embedded in the solution.

- The overall form of the solution is this: check the starting values to verify they are valid, then iterate through the valid range of values searching for the largest one. We save both the largest value as well as the address in scratch RAM of that value. It's that straight-forward.

- The first part of the solution verifies that the boundaries of the search are valid. For this, we assume that the starting address must be less than the ending address. If the starting and ending values are equal, or if the starting value is greater than the ending value, the subroutine immediately returns; otherwise, the solution assumes the address in r25 (the starting address) is less than value in r26 (the ending address). We're making an assumption here that this is what the problem wanted; note that the problem did not state exactly what to do if the value in r26 is less than the value in r25. The overall approach of this part is to ensure that what we're attempting to do is valid in the context of the remainder of the solution. In other words, if the value in r25 was greater than the value in r26 and the subroutine did not exit, the results would be strange and dangerous.

- The code associated with the "init" label grabs the first address as well as the value at that address. In this way, we assume that this is the "largest value"; we then check this value against the next location in scratch RAM. The instructions on lines 38-39 are interesting because the use both the address and value at that address in scratch RAM using direct and indirect addressing.

- The code at the "cont" label grabs the next value and address in scratch RAM. We do this by first incrementing the address then using that value to indirectly read the value at that address from scratch RAM.

- Once we have to scratch RAM values, we compare these values on line 45. There are two cases: the starting value is greater than or equal to the new value, or the starting value is less than the new value. In the latter case, we must replace the starting value with the new value; in the former case, we simply continue the algorithm by looking at the next address in scratch RAM. Lines 48-49 perform the writing of the new values.

- The code at the "check" label (line 51) checks to see if we've iterated through the entire range of values requested by the original starting and ending values. When we are done, we restore the value in register r25, which the solution uses as a working register, and then we exit the subroutine.

Example 15-40:

Write a RAT assembly subroutine that determines the parity of the value in a given memory location. The memory location is specified by the address in register r31. The subroutine returns the parity of the value in C flag with '1' and '0' indicating odd and even parity respectively. Make sure you subroutine does not alter any value in any memory location or register.

Solution:

Figure 15-47 shows a solution to Example 15-40. Fun notes follow a bit later.

```
(00)  ;-----------------------------------------------------------------
(01)  ;- subroutine: Parity
(02)  ;-
(03)  ;- This subroutine determines the parity of the value in a given
(04)  ;- memory location. The memory location is specified by the address
(05)  ;- in register r31. The subroutine returns the parity of the value
(06)  ;- in C flag with '1' and '0' indicating odd and even parity,
(07)  ;- respectively.
(08)  ;-
(09)  ;- Affected regs & flags:, C, Z
(10)  ;-----------------------------------------------------------------
(11)  Parity:
(12)  init:       PUSH    r0              ; save registers
(13)              PUSH    r1
(14)              PUSH    r2
(15)
(16)              LD      r0,(r31)        ; get the value
(17)              MOV     r1,0x08         ; init the loop count
(18)              MOV     r2,0x00         ; init the accumulator
(19)
(20)  loop:       LSL     r0              ; shift value into carry
(21)              ADDC    r2,0x00         ; add bit val to accumulator
(22)              SUB     r1,0x01         ; decrement loop count
(23)              BRNE    loop            ; branch to do more
(24)
(25)              AND     r2,0x01         ; mask LSB
(26)              BREQ    even            ; branch to clear carry
(27)              SEC                     ; odd case; set carry
(28)              BRN     done            ; branch to exit subroutine
(29)  even:       CLC                     ; even case; clear carry
(30)
(31)  done:       POP     r2              ; restore registers
(32)              POP     r1
(33)              POP     r0
(34)
(35)              RET                     ; go on home...
(36)  ; ----------------------------------------------------------------
```

Figure 15-47: The solution for Example 15-40.

Notes Regarding the Solution: This is another problem that uses many useful assembly language programming techniques and tricks. The notion of parity is important in digital-land, so eventually, you're going to have to generate parity associated with some set of bits. This problem asks you to generate parity associated with the data at a memory location specified indirectly by register r31. Recall that parity is an attribute of a set of bits where the bits have odd parity if there is an odd number of bits in the set that are '1'; the bits otherwise have even parity. Finally, here is some other fun stuff embedded in the solution.

- The program starts with an init section, which initially covers saving the three registers that the subroutine uses. The issue here is that these three lines of code will probably be some of the final code you write for this problem because you probably won't know what registers your solution will use until you're done with the problem. The three instructions (lines 12-14) save the registers by pushing them on the stack. The corresponding code (lines 31-33) pop these saved registers off the stack. Recall that the stack is a LIFO, so the popping order is opposite of the pushing order, which ensures the correct data ended up in the correct registers.

- Lines 16-18 show the next section of initialization code. First, we retrieve the value we're checking parity for from the scratch RAM, which we do with an indirect load instruction. Next, we initialize a loop count to eight, as there are eight bits we need to examine. Lastly, we initialize a register to hold the value of the "1 bits" we'll be counting. We refer to this register as our "accumulator" register.

- The algorithm we'll use is to shift a bit into the carry flag using the LSL instruction on line 20. This shifts either a '1' or a '0' into the carry. We follow with a reg-immed ADDC instruction with the source

operand set to 0x00. This effectively adds the carry bit to the accumulator register. We do this eight times as controlled by the loop iteration control construct on lines 22-23.

- When we exit the iterative loop, register r2 holds the number of one bits. The LSB of register r2 thus indicates the parity of the value in question as if the LSB is set, the value in register r2 is odd; otherwise, it is even. This being the case, we mask the LSB and use an if/else construct to either set or clear the carry flag for odd and even parity, respectively.

While the solution in Figure 15-47 is OK, we can do a bit better. Figure 15-48 provides a more clever solution. The key to the more clever solution is realizing that in order to place the correct value into the carry flag, all we need to do is perform a shift operation such as LSR (line 25 in Figure 15-48). This produces a solution with four less instructions than the solution in Figure 15-47. Having four less instructions is good because it uses less code space and the subroutine has a shorter run-time. These are two qualities you always strive for in assembly language programming.

```
(00)  ;-------------------------------------------------------------------
(01)  ;- subroutine: Parity
(02)  ;-
(03)  ;- This subroutine determines the parity of the value in a given
(04)  ;- memory location. The memory location is specified by the address
(05)  ;- in register r31. The subroutine returns the parity of the value
(06)  ;- in C flag with '1' and '0' indicating odd and even parity,
(07)  ;- respectively.
(08)  ;-
(09)  ;- Affected regs & flags:, C, Z
(10)  ;-------------------------------------------------------------------
(11)  Parity:
(12)  init:       PUSH    r0              ; save registers
(13)              PUSH    r1
(14)              PUSH    r2
(15)
(16)              LD      r0,(r31)        ; get the value
(17)              MOV     r1,0x08         ; init the loop count
(18)              MOV     r2,0x00         ; init the accumulator
(19)
(20)  loop:       LSL     r0              ; shift value into carry
(21)              ADDC    r2,0x00         ; add bit val to accumulator
(22)              SUB     r1,0x01         ; decrement loop count
(23)              BRNE    loop            ; branch to do more
(24)
(25)              LSR     r2              ; shift LSB into carry
(26)
(27)              POP     r2              ; restore registers
(28)              POP     r1
(29)              POP     r0
(30)
(31)              RET                     ; go on home...
(32)  ; -------------------------------------------------------------------
```

Figure 15-48: The more "clever" solution for Example 15-40.

Example 15-41:

Write a RAT assembly subroutine that swaps the values in two registers. For this problem, do not change the value in any other register or any memory location. The two registers in question for this problem are r30 and r31.

Solution:

```
(00)  ;-----------------------------------------------------------------
(01)  ;- subroutine: Reg_swap
(02)  ;-
(03)  ;- This RAT assembly subroutine swaps the values in two registers
(04)  ;- without changing the value in any other register or any memory
(05)  ;- location. The two registers in question for this subroutine are
(06)  ;- r30 and r31.
(07)  ;-
(08)  ;- Affected regs & flags: C, Z
(09)  ;-----------------------------------------------------------------
(10)  Reg_swap:   EXOR    r30,r31         ; It's a trick;
(11)              EXOR    r31,r30         ;  What can you say?
(12)              EXOR    r30,r31         ;
(13)
(14)              RET                     ; Home is calling...
(15)  ; ---------------------------------------------------------------
```

Figure 15-49: Solution forExample 15-41.

Notes Regarding the Solution: You sometimes see this problem in interviews, so it is somewhat of an important problem. It's completely non-intuitive; it's instead one of those problems you make yourself aware of and try to remember it when someone you're trying to impress asks you. You should remember this problems as the "EXOR trick" that uses three instructions. Beyond that, there's only one path to the solution, so it is actually easy to figure out in a pinch. Finally, is there more to say about this solution?

- There is not much to say about this solution, as it is a well-known trick. Run this code through a simulator or work an example to convince yourself it really works. And when you're a hotshot engineer in charge of interviewing perspective employees, be sure to ask them this question.

Example 15-42:

Write a RAT assembly language subroutine that sort the values in contiguous memory locations. Registers r30 and r31 hold the starting and ending memory addresses for this sort, where the address in r30 is at least two less than the address in r31. The result of this sort should place the largest value in memory location r30 and the smallest in r31. Don't alter the values in registers r30 and r31.

Solution: Figure 15-50 shows a solution to Example 15-42. Some interesting commentary follows the solution, so start looking forward to it.

```
(00)  ;-----------------------------------------------------------------------
(01)  ;- Subroutine: Mem_sort
(02)  ;
(03)  ;- Description: This subroutine sorts the values in contiguous memory
(04)  ;- locations. Registers r30 and r31 hold the starting and ending memory
(05)  ;- addresses for this sort, where the address in r30 is at least two
(06)  ;- less than the address in r31. The result of this sort should place
(07)  ;- the largest value in memory location r30 and the smallest in r31.
(08)  ;-
(09)  ;- Affected regs & flags: r5, r6, r7, r10, r11, r20, r21, C, Z
(10)  ;-----------------------------------------------------------------------
(11)  Mem_sort:  MOV    r10,r30         ; grab mem location values
(12)             MOV    r11,r31
(13)             SUB    r11,r10         ; calculate iteration count
(14)             MOV    r5,r11          ; copy loop count
(15)             SUB    r5,0x01
(16)             MOV    r7,r5           ; copy loop count
(17)
(18)  out_loop:  MOV    r6,r5           ; set inside loop count
(19)             MOV    r20,r30         ; grab mem location values
(20)             MOV    r21,r20         ; grab mem location values
(21)             ADD    r21,0x01        ; offset mem location by 1
(22)
(23)  in_loop:   LD     r0,(r20)        ; get mem value
(24)             LD     r1,(r21)        ; get mem value
(25)             CALL   Swap            ; call sort routine
(26)             ST     r0,(r20)        ; store sorted value
(27)             ST     r1,(r21)        ; store sorted value
(28)
(29)             ADD    r20,0x01        ; increment mem location index
(30)             ADD    r21,0x01        ; increment mem location index
(31)
(32)  in_cont:   SUB    r6,0x01          ; iteration control of inner loop
(33)             BRNE   in_loop
(34)
(35)
(36)  out_cont:  SUB    r7,0x01          ; iteration control of outer loop
(37)             BRNE   out_loop
(38)
(39)             RET                     ; come on up to the house
(40)  ;-----------------------------------------------------------------------
(41)
(42)  ;-----------------------------------------------------------------------
(43)  ;- Subroutine: swap
(44)  ;
(45)  ;- Description: this subroutine sorts the values in r0 and r1 into
(46)  ;- descending order (r0 will have the larger value).
(47)  ;-
(48)  ;- Tweaked registers: r0,r1
(49)  ;-----------------------------------------------------------------------
(50)  Swap:      CMP     r0,r1
(51)             BRCC    done
(52)             EXOR    r0,r1        ; tricky swap routine
(53)             EXOR    r1,r0
(54)             EXOR    r0,r1
(55)  done:      RET
(56)  ;-----------------------------------------------------------------------
```

Figure 15-50: A solution to Example 15-42.

Notes Regarding the Solution: This potentially a very useful problem in that it is one you may actually face out there in the real world. Here is that interesting commentary as previously promised.

- One nice thing about this problem and subsequent solution is that it is very generic. By this we mean that it will work on any given address range. The only thing that this solution does not check is to make sure that the memory location specified in register r30 is truly not at least two less than the address specified in register r31. Adding this feature would be a great form of error checking and ensure the solution is bulletproof. We don't do it here in order to save space in the solution.

- The solution in general is another bubble sort. The solution contains a nested loop. The inner loop sorts one pass of the given memory range; this sort occurs one less than the number of memory locations in the given range. The inner loop represents one pass of a "bubble-up". However, we must allow the value to bubble all the way up, so we must also execute the outside loop also by one less than the number of memory locations in the range.

- The first part of the solution (lines 11-16) is the initialization code. We save the sent values, calculate the number of memory locations to sort, then calculate the number of iterations we need to perform in order to arrive at the solution.

- We know this algorithm does a sort, so we write a "sort" subroutine. We'll be calling this subroutine multiple times from the main subroutine. This is a key step in the solution because using this subroutine makes the solution much cleaner. Any time you have a problem such as this one, the first thing you should do is put some time into architecting the solution, which means organizing your solution in such a way as to make solving the problem much more straight-forward.

- The "Swap" subroutine starting on line 50 provides the heart of the sort. This subroutine sorts the sent values in registers r0 & r1. This sort routine happens to sort the values in descending order. You could quickly modify this subroutine to sort in ascending order.

- The code associated with the "out_loop" label (lines 18-21) is actually setting up the iteration values for the inner loop. This consists of resetting the registers to point at the first two memory locations and refreshing the original loop count.

- The code associated with the "in_label" (line 23-30) transfers data from memory to the registers required by the "Swap" subroutine, calls the "Swap" subroutine, then transfers the data back to memory in sorted order. The code then updates the memory addresses (by incrementing them). The code associated with the "in_cont" label (lines 32-33) checks to see if the algorithm requires more iterations of the loop.

- The code associated with the "out_cont" label (lines 36-37) check to see if the outer loop requires more iterations.

Example 15-43:

Write a RAT assembly language subroutine counts the number of odd and even numbers in the first half of the scratch RAM. For this problem, consider all the values in the first half of scratch RAM to be unsigned integers. The subroutine then stores the number of even values in register r10 and the number of odd values in register r11. For this problem, make sure your assembly code solves this program in an efficient manner. Do not change any register other than r10 and r11.

Solution: Figure 15-51 provides a solution to Example 15-43. Some interesting commentary follows the solution, so start making your list and checking it twice.

```
(00)  ;-------------------------------------------------------------
(01)  ;- Subroutine: Count_vals
(02)  ;-
(03)  ;- This RAT assembly language subroutine counts the number of
(04)  ;- odd and even numbers in the first half of the scratch RAM (all
(05)  ;- considered to be unsigned 8-bit integers). The subroutine
(06)  ;- then stores the number of even values in register r10 and
(07)  ;- the number of odd values in register r11.
(08)  ;-
(09)  ;- Affected regs & flags: r10, r11, C, Z
(10)  ;-------------------------------------------------------------
(11)  Count_vals:
(12)  init_1:     PUSH   r0             ; save registers
(13)              PUSH   r1             ; r0 = iterative count
(14)                                    ; r1 = workding register
(15)                                    ; r11 = odd accumulator
(16)              MOV    r0,0x7F        ; count of half of memory
(17)              MOV    r11,0x00       ; clear accumulator
(18)
(19)  loop_1:     LD     r1,(r0)        ; load memory value
(20)              LSR    r1             ; shift LSB to Carry
(21)              ADDC   r11,0x00       ; increment r11 by C value
(22)
(23)              SUB    r0,0x01        ; decrement iterative count
(24)              CMP    r0,0x00        ; see if there is more to do
(25)              BRCC   loop_1         ; continue with loop
(26)
(27)  done:       MOV    r10,0x80       ; move 128 to r10
(28)              SUB    r10,r11        ; even count is 128-odd
(29)
(30)              POP    r1             ; restore saved values
(31)              POP    r0
(32)
(33)              RET                   ; come on up to the house
(34)  ;-------------------------------------------------------------
```

Figure 15-51: Solution for Example 15-43.

Notes Regarding the Solution: This is a classic problem that looks at a block of RAM (specified by two registers) and generates a piece of information regarding the memory block. This particular problem happens to count the number of odd and even values in the first half of RAM. Not too exciting but somewhat interesting if you're a boring programmer person. Here are the highlights of this rather interesting solution.

- The problem asks you two interesting things. First, it asks that you make you code efficient. This request generally implies that you'll need to do things in a generic manner, which for relatively straight-forward problems such as this one means use iterative constructs (loops) and indirect memory addressing. The second thing this problem asks for is to not change any registers other than r10 & r11. Requests such as this one generally mean that you must push the registers your solution uses onto the stack before you start your solution and pop them off the stack after you complete the solution.

- The solution uses two registers: r0 serves as an iteration counter that the solution sets to 0x7F (127) on line 16. This provides an index that we'll use along with indirect memory reads using the LD instruction on line 19. Keep in mind that the range of [0,127] represents 128 different numbers, which is what this solution requests. The solution uses register r1 as a working register; in particular, the solution loads values from memory into this register before it manipulates them.

- The key to this solution is found on lines 20-21. The solution left-shifts the memory value, which moves the LSB into the carry flag. The LSB is set when the value is odd and cleared when the value is even. The solution then adds the carry flag to the register r11. The solution initially cleared register r11 on line 17, thus r11 represents the count of negative numbers. If the number is even, register r11 does not change.

- When the solution breaks out of the iterative construct, we must calculate the even count. Because register r11 hold the odd count, we find the even count by subtracting the odd count from 128.

- As advertised, we must protect the registers that the solution changes. We don't care about registers r10 and r11, but we do need to protect registers r0 and r1. We do so by pushing these registers onto the stack at the beginning of the subroutine and popping them off before the subroutine returns. Generally speaking, the best approach to handing the protection of register in your subroutine is to first write your subroutine/solution; when you're done, go back and add the pushes and pops required to protect the register your solution uses.

15.5 Chapter Summary

- This chapter contained many example programs that show many common techniques to assembly language programming. Yes, lots of happy stuff embedded in those many solutions.

15.6 Chapter Exercises

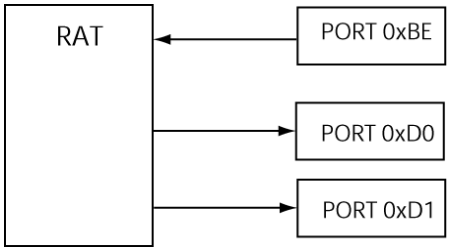
1) Write some C code that would generate the following RAT assembly code fragment. Include variable declarations for the associated C code variables and make them consistent with the following code.

```
            MOV     r1,0x1A

loop1:      ADD     r20,0x02
            SUB     r21,0x01

            SUB     r1,0x01
            BRNE    loop1
```

2) What is the value in r21 at the unconditional branch instruction?

```
main:       MOV     r20,0x22
            CALL    my_sub
            BRN     main        ; what is value of r21 here?


my_sub:     CMP     r20,0x11
            BRNE    v_neq

            MOV     r21,0x7A
            SUB     r21,0x01
            BRN     done

v_neq:      MOV     r21,0x89
            ADD     r21,0x01

done:       RET
```

3) Write a RAT assembly language program that inputs a value from input port 0xBE. If the input value is equal to 0xC4, complement the value and output the result to output port 0xD0; otherwise output the unchanged value to output port 0xD1. The program should perform this operation repeatedly.



4) Write a RAT assembly language program that does the following: If both SW(3) and SW(2) are on (an on switch has a value of '1'), then turn on LED(7) and LED(6) (the other LEDs are not affected); otherwise toggle LED(1) and LED(0) (the other LEDS are not affected). The program does this continuously.

- Assume there are eight switches connected to PORT 0x30: SW(7) → SW(0)
- Assume there are eight LEDs connected to PORT 0x0C
- Assume that r20 holds the current state of the LEDs

5) State the two conditions that make the following two subroutines equivalent.

```
; ------------------------------------------------
my_sub1:    PUSH   r30
            MOV    r31,0x88
            ST     r31,(r12)
            RET
;------------------------------------------------
;------------------------------------------------
my_sub2:    PUSH   r30
            MOV    r31,r9
            ST     r31,0x72
            RET
;------------------------------------------------
```

6) Write a RAT assembly language subroutine that counts the number of bits that are set in register r30 and stores the value in register r31. The subroutine should not alter the contents of r30. Don't use more than twelve instructions in your solution.

7) Briefly state what the following code fragment is doing; then write a *minimum* number of lines of code that does the same thing without using PUSH & POP instructions. There is no restriction on the number of registers you can use for this problem.

```
; ----------------------------
frag:       PUSH   r2
            PUSH   r3
            POP    r3
            POP    r2
;----------------------------
```

8) Write a RAT assembly language subroutine that modifies a value in scratch RAM. The scratch RAM location in question is represented by the value in r20 (the value in r20 is interpreted as an address). If the value at that location is even, then multiply the value four and store the result back at that scratch RAM address; otherwise, divide the value by two and store the value back at that scratch RAM address. Don't worry about overflow and underflow for this problem.

9) Write a subroutine that inputs a value from port 0xCC, transforms the value according to the following equation: out=1.5(in) + 7, and outputs the result to port CD. Round up the value calculated using the equation. Make sure the range of output values is: [19,252]. Provide flowchart for your solution.

10) Write a RAT assembly language subroutine that sort the values in contiguous memory locations. Registers r30 and r31 hold the starting and ending memory addresses for this sort, where the address in r30 is at least two less than the address in r31. The result of this sort should place the largest value in memory location r30 and the smallest in r31. Don't alter the values in registers r30 and r31.

11) Write a RAT MCU assembly language subroutine that copies values from one range in scratch RAM to another range. The range is given by the addresses in [r0,r1]; copy these values to the range starting at the address in r10. If the subroutine copies two values of 0xFF, set the C flag and exit the subroutine; otherwise, complete the copying and clear the C flag.

- Assume: r0 <r1<r10; (r10+(r1-r2) < 255)
- Don't allow the subroutine to permanently change any register
- Minimize the number of instructions in your solution

12) Write an interrupt driven RAT MCU assembly language program that outputs the most recent value on the switches to the LEDS when it receives an interrupt. If the same switch value is output on two consecutive interrupts, the program stops outputting values until BUTTON(0) is pressed, at which point it returns to normal processing.

- Don't use an IN instruction in the interrupt service routine
- There same switch values will never appear more than two consecutive interrupts
- Assume an external device generates the interrupt
- Minimize the number of instructions in your solution

```
.EQU    LEDS     = 0x44
.EQU    SWITCHES = 0x45
.EQU    BUTTONS  = 0x46
```

16 The RAT Assembler

16.1 Introduction

The assembler represents the programmer's interface between writing an assembly language program and converting that program to a format that the computer hardware can understand. The assembler's main task is to convert assembly language source code into machine code. Beyond that, you'll find that different assemblers have a different amount features, many of help programmers write code that is both functional and easy for other human programmers to understand.

This chapter describes the RAT assembler and its modest feature set. The information in this chapter goes beyond what the assembler manual presents in the level of detail and explanation it provides.

Main Chapter Topics

AUTOMATIC FILE GENERATION: This chapter describes the various files that the RAT assembler automatically generates.

MEMORY SEGMENTATION: This chapter describes how the RAT assembler models the RAT MCU memory usage including the program and scratch memory.

ASSEMBLER DIRECTIVES: This chapter describes the various assembler directives available to the RAT assembly language programmer.

Why This Chapter is Important

This chapter is important because it describes all aspects of the RAT assembler and how the RAT assembly language programmer can properly utilize them.

16.2 RAT Assembler Overview

The RAT assembler is responsible for generating machine code for the RAT instruction set. In particular, the RAT assembler generates a VHDL model that users can use to synthesize a memory to use as the program memory in a RAT MCU model. The machine code associated with the assembly program is coded within the body of the VHDL model of the program memory. The final program memory is officially read-only; we forever refer to it as the "prog_rom".

The RAT assembler, Ratasm, was written in a CYGWIN environment and is based on standard scanning (FLEX) and parsing (YACC) tools. The RAT assembler is a two-pass assembler; the first pass primarily locates various labels and assigns them appropriate values; the second pass assigns assembly instruction bits (opcodes and field codes) and inserts startup code as required.

16.3 Assembler Automatic File Generation

The RAT assembler, ratasm, is specifically designed for the beginning assembly language programmer in mind. As a result, ratasm includes in-depth error checking and report generating. Running ratasm generates several files, which will aid in program understanding and/or debugging. This section lists the files generated by ratasm. For all the assembler generated files, ratasm replaces the "xxx" prefix with the filename (not including the .asm extension) of the program that ratasm attempted to assemble. All RAT assembly programs must include an ".asm" extension in order for ratasm to recognize and possibly assemble the file.

16.3.1 The Assembly Language Listing File: xxx.asl

This is the output list file associated with the assembly language program that was successfully assembled. The assembler automatically generates this file when the assembler attempts to assemble the given source code. The assembler generates this file independent of whether the assembler is able to successfully assemble the source code or not. If there are errors in the source code, the xxx.asl file will mark the location of the error and possibly provide a helpful message regarding the error.

The assembly language listing file provides all associated instruction opcodes, assembler directives, data memory information, symbol table listing, and other useful information that could possibly be of use to the programmer. If your assembly language program contains errors that you are not able to figure out by other means, examining the assembly language listing file could be helpful. If there is any information missing from the assembly language listing file, you probably don't need it.

16.3.2 The Assembly Language Error File: xxx.err

If the program you are attempting to compile contains errors, the xxx.err file provides a list of those errors and associated error messages. There are two types of errors in the ratasm assembler: 1) errors that the assembler can pinpoint, and 2) errors that the assembler knows are errors but cannot specify what the exact error is. In the first case, ratasm prints a relatively helpful error message to help the programmer find and fix the error. In the second case, however, ratasm provides little or no information about the error other than the line in the program where ratasm assumes the error occurred.

As will most assemblers, and compilers for that matter, often the first error is the most important error. This is because once the assembler encounters one error, it often times becomes out-of-sync as it attempts to assembler the remainder of the source code. This is one of those issue where the assembler may tell you there are many errors, but all the error disappear after you repair the first error.

16.3.3 The Machine Code File: "prog_rom.vhd"

This file is a VHDL model of a ROM object containing the machine code for the successfully assembled program. The model is actually some type of RAM that has the write input disabled. The RAT assembler only generates this file when the assembly language program successfully assembles. If the program does not successfully assemble, the assembler deletes any existing prog_rom.vhd file in the directory where you are running the assembler.

16.3.4 The Debug File: xxx.dbg

This is a specially formatted file containing information used by the debugger/simulator. If you're not the debugger, then you won't need the information contained in this file.

16.4 RAT Assembly Language Overview

Assembly language programs have four distinct parts: 1) comments, 2) labels, 3) assembler directives, and 4) assembly language instructions. Later sections address RAT assembler directives and instructions.

16.4.1 RAT Assembler Comments

The RAT assembler uses the semi-colon character (;) to indicate a comment. The comment character can appear in any column of source code text. The RAT assembly considers all text following the comment character on any source code line to be a comment.

16.4.2 RAT Assembler Labels

The ratasm assembler uses labels to mark locations in both the code and data segments. Labels are specified by a string followed immediately (without white space) by a colon. Label definitions in the code segment can appear as the first field in any valid instruction or can appear on lines by themselves. Label definitions in the data segment can appear without associated assembler directives. Label definitions can appear on associated lines with some directives, but not all of them (see the section on assembler directives). Multiple label definitions cannot appear on the same line in the source code. There is no limit to the number of labels you can use in a program.

The assembler assigns all labels a value that is associated with the segment the label appears in. The notion of labels in the code segment is relatively simple. The assembler contains a counter that is initialized to the argument of the .ORG assembler directive. When the assembler encounters a label, it assigns the label the value of the current counter. When the assembler encounters an instructions, the assembler increments the value of the counter. Figure 16-1 shows an example of various label usage. The important thing to note from Figure 16-1 is that labels do not increment the segment counter; thus, lab1, lab2, lab3 and lab4 all have the same value of 0x55.

```
;-------------------------------------------------------------------------
;-- Example use of labels
;-------------------------------------------------------------------------
.CSEG
.ORG    0x55

lab1:
lab2:
lab3:
lab4:   LSL    R5     ; instruction at address 0x34
        LSR    R6     ; instruction at address 0x35

;-----------------------------------------------------------
;- Subroutine: my_sub (a label for a subroutine)
;- Description: This is an example showing the subroutine
;-  calling/returning mechanism; it does nothing useful.
;-
;- Registers modified: r1, r9
;-----------------------------------------------------------
my_sub:   AND   r1, r2        ; some random operation
          OR    r9, r3        ; more stuff
          ...                 ; more stuff
          RET                 ; return to calling program
;-----------------------------------------------------------
```

Figure 16-1: Example of various approaches to using labels.

The notion of using labels in the data segment is a bit more complicated. Some labels in the data segment allow programmers to “set aside” multiple bytes of storage per line of code. In this case, the assembler increments the

counter associated with the data segment after it encounters each byte of data. Please see the example of this later in this chapter.

The assembly language source code requires labels for program flow control-type instructions to work properly. Outside of these requirements, programmers should use labels as a form of self-commenting. In particular, programmers should use labels to delineate sections of code with meaningful names. Note that labels do not increase overall program size as labels are not the same thing as or part of instructions.

16.5 RAT Memory Segmentation

The RAT MCU architecture comprises of the many modules including various memories, a program counter, I/O, interrupts, a control unit, and various other control hardware. The RAT assembler has direct involvement with two types of memory in the RAT architecture: the program memory and the scratch RAM. The assembler generates and assigns machine code to the program memory and also generates the associated VHDL model. Additionally, the assembler is responsible for assigning values to the scratch RAM. In particular, the when the programmers requests that particular memory locations be initialized, the assembler automatically generates what we refer to as "start-up code", which involves generated a pair of MOV and LD instructions and inserting those instructions before the programmer's program begins execution. We'll describe this in more details later.

Computer hardware is generally comprised of different memory modules, each serving a distinct purpose. Software items such as assemblers generally have options to configure these memories prior to actual program execution. Some microcontrollers are quite complex, which makes writing source code for them relatively complicated. The RAT MCU is relatively simple, and thus does not have as many options or features as do more complex microcontroller.

The programmer's view of the RAT MCU models the two memory types as program memory and scratch RAM. The RAT assembler allows the programmer several configuration options and controls for these memories. The RAT assembler models these memory types as one memory that is divided into two segments: the code segment and the data segment. The code segment refers to instruction memory while the data segment refers to scratch RAM. Because of these two memories, the RAT assembler requires the programmer to explicitly specify which segment a particular instruction or directive belongs to using assembler directives. The RAT assembler contains commands we refer to as "assembler directives" that allow programmers to configure these memories with a modest set of controls. The .DSEG and .CSEG assembler directives specify the data segment and code segment, respectively. We'll explain these topics in full in the following section under the sections of the appropriate assembler directives.

16.6 RAT Assembler Directives

The ratasm assembler has several directives in order to provide the programmer with more versatility in overall program design. The directives enforce a clear and concise style of programming and generate errors and warnings upon misuse. Table 16.1 shows the list of ratasm directives; an explanation of these directives follows the table.

Directive	Short Description
.CSEG	Indicates following information is associated with code segment
.DSEG	Indicates following information is associated with data segment
.ORG	Allows to adjust information placement in code and data segments
.EQU	Allows numeric values to be associated with strings
.DEF	Allows register file register to be associate with string
.BYTE	Allows user to reserve uninitialized memory
.DB	Allows user to reserve and initialize memory

Table 16.1: A list of RATasm assembler directives.

16.6.1 Segment Directives

We effectively model the RAT MCU's memory space as being in one of two segments: the code segment or the data segment. The .CSEG and .DSEG directives provide a means to differentiate between code and data segments. The assembler places opcodes of all program instructions in program memory and thus forms the code segment. All declared data is associated with data memory and is thus part of the data segment. Executable instructions must appear in the code segment while memory-type directives must appear in the data segment. Further details appear in the following sections. The RAT assembler generates an error if you attempt to use a particular directive in the incorrect segment.

16.6.1.1 Directive: .CSEG

Programmers use the .CSEG directive to indicate that all the labels after the .CSEG directive are defined in terms program memory (as opposed to data memory). Instructions can only appear in the code segment while data declarations can only appear in the data segment. The code segment is associated with program memory.

The .CSEG directive has no argument. When programmer use the .CSEG directive, the code memory address reverts to the either the code memory position one location after the previously issued instruction (in the case where the directive does not follow an .ORG directive) or reverts back to the previously issued .ORG value. The .CSEG directive must appear in the first column of the source listing and thus cannot have a label on the same line. Figure 16-2 shows an example of .CSEG usage.

16.6.1.2 Directive: .DSEG

Programmers use the .DSEG directive to indicate that all the labels after the .DSEG directive are defined in terms program memory (as opposed to data memory). Instructions can only appear in the code segment while data declarations and initializations can appear in the data segment. Attempts to declare instructions in the data segment will result in an assembler error.

The .DSEG directive requires no argument. When programmer uses the .DSEG directive, the data memory address reverts back to the either the data memory position one location after the previously declared memory (in the case where the directive does not follow an .ORG directive) or reverts back to the previously issued .ORG value that was issued in the data segment. The .DSEG directive must appear in the first column of the source listing and thus cannot have a label on the same line. Figure 16-2 shows an example of .DSEG usage.

16.6.2 Directive: .ORG

The .ORG directive is shorthand for "origin". Programmers use this directive to place data and instructions at known locations in either the data or the code segments, respectively. The .ORG directive takes one argument, which sets the location counter in the given segment. Both the code and data sections maintain their own counters, which increment according to the data declarations (the data segment) or listed instructions (the code segment). The .ORG directive effectively sets these respective counters to the value associated with the argument provided with the .ORG directive. The .ORG directive must appear in the first column of the source listing and thus cannot have a label on the same line. Figure 16-2 shows an example of the .ORG directive in both the code and data segments.

```
;-------------------------------------------------------------------------
;- code fragment
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;-- .ORG used in data segment
;-------------------------------------------------------------------------
.DSEG
.ORG   0x5A    ; set the data segment counter to 0x5A

var_name1  .BYTE    0    ; associated var_name1 with memory address 0x5A
var_name2  .BYTE    0    ; associated var_name2 with memory address 0x5B

;-------------------------------------------------------------------------
;-- .ORG used in code segment
;-------------------------------------------------------------------------
.CSEG
.ORG   0x34    ; sets the code segment counter to 0x34

instr1:    LSL    R5     ; instruction at address 0x34 (label instr1=0x34)
instr2:    LSR    R6     ; instruction at address 0x35 (label instr2=0x35)
```

Figure 16-2: Example usage of the .ORG directives in both code and data segments.

16.6.3 Directive: .DB

Programmers use the .DB directive to reserve and initialize a given number of bytes in scratch memory starting at the current data memory address (the address associated with the data segment counter). Programmers can use this directive either with or without a label on the same line. When the .DB directive appears with a label on the same line, the assembler assigns the label the current data memory counter and allows it for use in some RAT instructions. When you use the .DB directive without a label, the assembler initializes memory to the current address of the data segment counter. The .DB directive initializes one byte for each decimal or hex number following the .DB directive. The internal data segment automatically tracks the proper location in data memory as the programmer specifies more memory locations. The .DB directive can only appear in the data segment.

```
;-------------------------------------------------------------------------
;- code fragment
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;-- .DB Directive usage
;-------------------------------------------------------------------------
.DSEG                    ; we're in the data segment
.ORG   0x20              ; set the data segment counter to 0x20

                  ; define ascii equivalent for numerals 0 to 9
                  ; 0x30 = '0', 0x31 = '1', etc.
                  ;
ascii_digits:     .DB 0x30, 0x31, 0x32, 0x33, 0x34      ; with label
                  .DB 0x35, 0x36, 0x37, 0x38, 0x39      ; without label
```

Figure 16-3: Example of the .DB directive.

Example 16-1:

For the following code fragment, list the numerical value that the assembler associates with each label in the code and provide the address location in scratch RAM that will store the value 0x33.

```
;-------------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0x40            ; set the data segment counter to 0x40
                     ; define ascii equivalent for numerals 0 to 9

ascii_stuff:
ascii_digits1:  .DB 0x30, 0x31, 0x32, 0x33, 0x34
ascii_digits2:  .DB 0x35, 0x36, 0x37, 0x38, 0x39
ascii_digits3:
;-------------------------------------------------------------------
```

Solution: The data starts at address 0x40 in data memory because of the .DSEG and .ORG directives. The data segment counter thus starts at 0x40 and increments each time it encounters a byte of data. This the address value for the "ascii_stuff" label is 0x40 as it has not seen any data yet. The address of the "ascii_digits1" label has also seen no data so the value of this label is also 0x40. There are five bytes of data defined before the code encounters the "ascii_digits2" label, so the "ascii_digits2" label has the value of five greater than 0x40, or 0x45. The "ascii_digits3" label encounter five more bytes of data after the "ascii_digits2" label or a total of ten bytes of data after the "ascii_digits1" label, which gives a value of 0x4A to the "ascii_digits3" label.

16.6.4 Directive: .BYTE

Programmers use the .BYTE directive to reserve a given number of uninitialized memory locations starting at the current data memory address. You can use this directive either with or without a label on the same line. When the .BYTE directive appears with a label, the assembler assigns the label the current data segment counter value and can appear in appropriate RAT instructions. The one argument to the .BYTE directive specifies the number of bytes to reserve in memory (uninitialized) starting at the current data memory location. The internal data segment counter automatically tracks the proper location in data memory as the programmer declares more memory locations. The .BYTE directive can only appear in the data segment. Figure 16-3 shows the two forms

of the .BYTE assembler directive. In the code of Figure 16-3, the assembler assigns the btn_cnt1 and btn_cnt2 labels the value of 0x30 and 0x36, respectively.

```
;-------------------------------------------------------------------------
;- code fragment
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;-- .BYTE Directive usage
;-------------------------------------------------------------------------
.DSEG                    ; we're in the data segment
.ORG   0x30              ; set the data segment counter to 0x30

btn_cnt1: .BYTE     6    ; declare 6 bytes of data starting at data
                         ;  memory address 0x30; the label can be
                         ;  used for accessing the specified data
btn_cnt2: .BYTE     3    ; declare 3 bytes of data starting at data
                         ;  memory address 0x36
          .BYTE     3    ; declare 3 bytes of data starting at data
                         ;  memory address 0x39
```

Figure 16-4: Example of the .BTYE directive.

16.6.5 Directive: .EQU

The .EQU directive stands for "equate" and associates a label with a numeric value, where the programmer can specify the value as being either a decimal or hexadecimal value (decimal values must use a "0x" prefix). This directive allows for the replacement of specialized values such as bit-masks with more descriptive alpha-type names, all of this in the name of self-commenting. The .EQU directive can appear in either the code or data segments. It is customary assembly language programming practice to place all .EQU assembler directives somewhere near the beginning of the assembly source code file and before any assembly language instruction. The .EQU directive requires a label value field and a numeric value field. To properly use this directive, you must place an equals sign must between these two fields. Figure 16-5 shows examples of the .EQU directive.

The .EQU directive is one of the more common directives programmers use in assembly language programs. The main advantages of using this directive is that it makes programs more readable to other humans by replacing numeric values with descriptive alpha values. Generally speaking, programmers use the .EQU directive to replace a numeric value with a mnemonic that gives a hint of how the program will use that value. Another distinct advantage of using .EQU directives is that they make you code more portable. In case a numeric value changes are a result of hardware changes, the programmer only needs to change the value in the .EQU directive rather than hunting down every occurrence of the number in the associated program.

```
;-------------------------------------------------------------------
;- Port Constants
;-------------------------------------------------------------------
.EQU ADC_PORT     = 0x02          ; port for ADC
.EQU ENG_PORT     = 0x0C          ; port for English converter
.EQU RAT_CLR_PORT = 34            ; port for clear all regs port
;-------------------------------------------------------------------

;-------------------------------------------------------------------
;- Bit Mask Constants
;-------------------------------------------------------------------
.EQU BTTN1_MASK = 0x08        ; mask all but BTN5
.EQU BTTN2_MASK = 0x02        ; mask all but BTN5
;-------------------------------------------------------------------
```

Figure 16-5: Examples of the .EQU assembler directive.

16.6.6 Directive: .DEF

The .DEF directive stands for "define" and associates a label with a register. This directive allows for the replacement of basic register specifiers, such as "r0", "r1", etc., with labels. The main purpose of this directive is to make reading and understanding RAT assembly language programs easier for the human reader. This directive is a message from the programmer to the assembler to interpret labels as registers. This being the case, programmers should strive to use this directive in a self-commenting manner only with a special designated prefix, such as "r_". Having the ability to specify labels in place of registers allows programmers to use more descriptive alpha-type names, which supports the notion of self-commenting labels. The .DEF directive can appear in either the code or data segments, though it is customary assembly language programming practice to place all .DEF assembler directives somewhere near the beginning of the assembly source code file and before any assembly language instruction. The .EQU directive requires a label value field and a register name separated by an equals sign, thus the programmer must place an equals sign must between these two fields. Figure 16-6 shows a few examples of the .DEF directive. Quite often, refer to the labels that represent registers from the .DEF directive as "aliases".

```
;-----------------------------------------------------------------
;- Register Aliases
;-----------------------------------------------------------------
.DEF R_COUNT     = r30          ; register used for iteration
.DEF R_OUT_VAL   = r10          ; register used to store output value
.EQU R_WORKING   = r0           ; working register
.EQU R_ACCUM     = r2           ; register used as accumulator
;-----------------------------------------------------------------
```

Figure 16-6: Examples of the .DEF assembler directive.

16.7 Start-up Code

As you know from reading the previous sections, there are two assembler directives that involve "reserving" data in the scratch RAM: the .BYTE and the .DB directive. Both of these directives allow programmer to set aside specific bytes in the scratch RAM of special usage by the program. While these directives differ in many ways, they have one big difference: the .DB directive can initialize memory while the .BYTE directive cannot.

Anyone who has programmed a computer knows that you generally must declare the memory you're using. Additionally, you can also initialize the memory you declare. When you're programming using a higher-level language, the notion is initializing memory is not something you think too much about; you "just do it". But we're at the point where we can understand the notion of initializing memory as we're dealing with the RAT MCU at many different levels. There is no magic troll that initializes memory in the RAT MCU, or any other computer for that manner. There is some entity somewhere along the way that must initialize the memory. The RAT MCU uses an approach that is common among microcontrollers; this section describes that approach with its description of start-up code.

When you use the .DB directive to define a single byte of data, you the programmer expect that data to be in the scratch RAM at the location you specified when your program starts running. The only way to get that data into scratch RAM is to place it there under program control. The catch here is that you don't need to write the code that places the data into scratch RAM: the assembler does it for you by generating and running what we refer to as "start-up" code. Here are the main issues with the start-up code.

- The start-up code consists of pairs of MOV and ST instruction. The notion here is that the only way you can get data into scratch RAM is through a register, so the MOV instruction places data into a register (a MOV immediate instruction) and the ST instruction transfers that data to scratch RAM at a given address. The result is that for every byte of data that you declare with the .DB directive, the assembler generates two lines of assembly code.

- You the programmer must now be careful if you use the .DB directive because the assembler attempts to place the start-up code such that it runs before any other code in your program. To ensure this really occurs, the assembler places the start-up code starting at address 0x000 in the program memory. Your responsibility as the programmer is to ensure there is enough space for the assembly to place the start-up code should your program use the .DB directive. This means that if you use the .DB directive to declare and initialize ten bytes of data, you need to place the start your program at program memory address 0x15 or greater. Recall you can use the .ORG directive to place the program at any location in scratch RAM.

Example 16-2:

In the following code fragment, what is the lowest address in program memory that you can start storing your program code?

```
;-------------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0x40            ; set the data segment counter to 0x40
                     ; define ascii equivalent for numerals 0 to 9

wank:
wow_man1:  .DB 0x10, 0x30, 0x31, 0x32, 0x33, 0x34
wow_man2:  .DB 0x20, 0x35, 0x36, 0x37, 0x38, 0x39
;-------------------------------------------------------------------
```

Solution: The code in Example 16-2 uses two .DB directives to declare and initialize 12 bytes of data. Each byte of data generates two assembly language instructions (MOV and ST) for the start-up code. This means your program cannot start at addresses 0x00→0x17 in program memory. The lowest address value you can start your program at is thus: 0x18 (or 24 decimal). Keep in mind that the first storage location in memory is 0x00.

16.8 Chapter Summary

- The assembler automatically generates a set of files that are used for different purposes. These file include the assembly language listing file, the error file, the prog_rom.vhd file, and the debug file.
- The assembly language listing file contains a complete listing of all the data, constants, op-codes, label values, and more. This file can be used to debug programs when all else fails.
- Assembly code programs are divided into three parts: the assembly language code, the comments, and the assembler directives. Comments are used to make the program clearer to human readers. Labels can also be forms of comments (the self-commenting principal) if the programmer chooses them judiciously.
- The RAT assembler models the memory as being a data segment and a code segment. The associated RAT assembly program must therefore declare which segment is being used, which it does by using the appropriate segment directives. The data segment is associated with the scratch RAM while the code segment is associated with the program memory.
- The RAT assembler contains various segment directives that the programmer can use to communicate with the assembler. Assembler directives are not instructions and thus their usage does not require program memory space.
- The start-up code is associated with the use of the .DB directive. The .DB directive is used to declare and initialize memory in the RAT assembler. Each byte of memory declared using the .DB directive causes the assembler to automatically generate the start-up code, which comprises of a MOV and ST instruction for each byte of memory. The programmer must allow code space for the start-up code by placing the program code with a high enough address such that the assembler can place any start-up code before it. Leaving space for the start-up code is the responsibility of the programmer; in-properly placed start-up code will cause indeterminate program operation.

16.9 Chapter Exercises

1) Briefly describe the main difference between the .BYTE and the .DB directive.

2) In your own words, state why using the .EQU directive can be a great use in a typical assembly language program.

3) For the following code fragment, list the numerical value that the assembler associates with all the labels in the code. Also list the location in scratch RAM that the value 0x1E is stored in after the program initializes the data.

```
;-------------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0x22

stuff1:
my_stuff1:  .DB   0x23, 0xA4, 0x23, 0xAF, 0xE9, 0x87

stuff2:
my_stuff2:  .DB   0x32, 0x01, 0x08, 0x1E
```

4) For the following code fragment, list numerical value that the assembler associates with all the labels appearing in the code. Also list the value of data that will be stored at location 0x6A in scratch RAM.

```
;-------------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0x60            ; set the data segment counter to 0x40
                     ; define ascii equivalent for numerals 0 to 9

ascii_stuff:
ascii_digits1:  .DB 0x30, 0x31, 0x32, 0x33, 0x34, 0x45, 0x46
ascii_digits2:  .DB 0x35, 0x36, 0x37, 0x38, 0x39, 0x4A, 0x4B

.CSEG
.ORG    ?????
```

5) For the following code fragment, list the numerical value that the assembler associates with all the labels appearing in the code. Also, list the location in scratch RAM that the value 0x88 is stored in after the program initializes the data.

```
;-------------------------------------------------------------
;-- .DB Directives
;-------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0x50

happy1:
happiness7:  .DB   0x15, 0xD4, 0xEE, 0xA0, 0x39, 0xF0, 0x27

happy2:      .DB   0x88
happy3:
happiness8:  .DB   0x12, 0x06, 0x05, 0x1F
```

6) For the following code fragment, what is the minimum value for "?????" to make the program happy?

```
;-----------------------------------------------------------------
;-- .DB Directives
;-----------------------------------------------------------------
.DSEG               ; we're in the data segment
.ORG 0x40

Fred:
Bob:   .DB 0x30, 0x31, 0x32, 0x33, 0x34, 0x45, 0x46, 0x12
Tom:   .DB 0x35, 0x36, 0x37, 0x38, 0x39, 0x4A, 0x4B, 0x3F

.CSEG
.ORG    ?????

Main:   BRN   Main
```

7) For the following code fragment, what is the minimum value for "?????" to make the program happy?

```
;-----------------------------------------------------------------
;-- .DB Directives
;-----------------------------------------------------------------
.DSEG               ; we're in the data segment
.ORG 0x10

Cat:
Pig:   .DB 0x30, 0x31, 0x32, 0x33
       .DB 0x35, 0x36, 0x37, 0x38, 0x39, 0x4A, 0x4B, 0x3F

.CSEG
.ORG    ?????

Main:   BRN   Main
```

8) For the following code fragment, what is the minimum value for "?????" to make the program happy?

```
;-----------------------------------------------------------------
;-- .DB Directives
;-----------------------------------------------------------------
.DSEG               ; we're in the data segment
.ORG 0x00

Armstwong:
Bytes:       .DB 0x30, 0x31, 0x32
Big_time:    .DB 0x36, 0x37, 0x38, 0x39, 0x4A, 0x4B, 0x3F

.CSEG
.ORG    ?????

Main:   BRN   Main
```

9) For the following code fragment, what is the minimum value for "?????" to make the program happy?

```
;-----------------------------------------------------------------
;-- .DB Directives
;-----------------------------------------------------------------
.DSEG                ; we're in the data segment
.ORG 0xB0

Ima:
Wankster:       .DB 0x30, 0x31, 0x32, 0x33, 0x34, 0x45, 0x46, 0x12, 0x34, 0x00
All_Day_Long:   .DB 0x35, 0x36, 0x37, 0x38, 0x39, 0x4A, 0x4B, 0x3F, 0x45

.CSEG
.ORG    ?????

Main:   BRN   Main
```

17 Assembly Language Program Design

17.1 Introduction

This set of notes introduces a standard structured approach to assembly language programming. As the programs you write become more complex, it becomes important for you to take a healthy and sane approach to designing and writing programs. The use of *flowcharts* has many advantages and should become an important initial step in any software project spend time on. The chapter examines the basics of flowcharts and briefly discusses the theory behind them. The underlying theme of this discussion is the production of modular assembly language code.

Main Chapter Topics

AN APPROACH TO WRITING ASSEMBLY LANGUAGE PROGRAMS: This chapter provides an outline of the appropriate approach to writing assembly language programs.

MOTIVATION FOR FLOWCHARTS: This chapter provides brief motivational verbage that highlights the advantages of using flowcharts.

FLOWCHARTING SYMBOLS: This chapter presents and describes the basic symbols associated with flowcharting.

FLOWCHARTING EXAMPLES: This chapter provides examples that show how to apply the basic flowcharting symbols.

Why This Chapter is Important

This chapter is important because it outlines how to approach assembly language programs and describes the advantages of using flowcharts in assembly language program design and documentation.

17.2 Problem Solving with Programming

In the rush to complete your assignment for your instructor of supervisor, you can easily lose track of what it is your attempting to accomplish. In the worst cases, you find yourself mired with either the low-level details while ignoring the big picture, or you completely grasp the big picture while being unaware of the important low-level details you're probably passing over. In the end, if you want to be a successful programmer, you need to answer "yes" to the following questions:

1) **Did you write your program in a reasonable amount of time?** If you answer "no", you need to realize that you can't spend forever writing the program… at some point you have to call it done.
2) **Does your program work properly in all possible cases?** Of course if you answer "yes" to this question, it means that you've course tested the crap out of your program
3) **Can someone else easily understand and/or resuse your code?** If you anwers "no" to this questions, then you're either a bad programmer or you're into creating job security for yourself.

The key to ensuring that you answer "yes" to all of these questions is to keep your programs a simple as possible. Why? Because complex programs, if they work at all, are well-known to be fragile. A fragile program is like an academic administator's ego: you constantly worry about breaking it if you do anything wrong. Yes, the program running on your smart phone is a result of millions of lines of code and is seemingly complex, but that's not the point. If even the most complex program can be decomposed into simple building blocks, then the program is by definition simple[1]. Once again, the key to writing good programs is writing simple, well-structured code.

After you've been programming in assembly for a while, you'll find that you've probably developed your own coding style and your own approach to the entire "problem solving" package. Recall that the reason you're writing any program is to solve some problem. When you first start out programming, particularly using an assembly language, you should take a nicely disciplined approach. This section describes a high-level approach to the entire problem solvoing process, not just the program writing part of the approach. Though this is certainly not the only approach you can take, it's the approach you should take until you have developed your own successful assembly language programming style.

There are two basic requirements you must meet before attempting to solve problems by writing assembly language programs: 1) Understand the instruction set, and, 2) understand basic programming constructs and techniques. These two items are somewhat detailed and we provide more information in the following verbage.

1) **Understand the Instruction Set:** What this means exactly is that you must understand every aspect of the instruction set if you plan to write viable problems. With a 50-instruction processor, such as the RAT MCU, this is totally possible particularly since none of the instructions is that complex[2]. Some of the aspects you need to be familiar with are:

   a. Peculiar aspects of individual instruction: For example, how the RAT MCU handle the C flag for the shift and rotate instructions

   b. How the various instructions affect the C & Z flags: Different instructions affect the condition flags differently.

2) **Understand Basic Programming Techniques and Constructs:** You must understand basic approaches to programming in order to write code. One of the many good things about assembly language programming is that there are only a few constructs that you need to know; even the most complicated assembly language program is a conglomeration of these constructs. The basic items we're referring to are:

   a. Iterative Constructs (Loops): The two types of iterative constructs are loops when you know in advance how many times you'll iterate (based on a count) and when you don't know how many times you'll iterate (based on a condition). Either of type of iterative constructs can be further classified as a "while loop" or a "do-while" loop.

   b. if/else Constructs: The if/else construct is the basic decision-making program flow construct in assembly language programming.

   c. Bit Masking: Bit masking is one of the basic techniques to operate on bits, which is generally what you'll find yourself doing on a microcontroller[3].

Once you've met the basic requirements, you're then ready to solve the problem by writing an assembly language program. The three basic steps to writing assembly language programs are:

[1] So when your phone freezes of does something stupid... probably bad code. Please don't blame the hardware.
[2] A more precise statement may be that you should understand all aspects of the instructions you plan on using. But because MCUs are designed to be usable, being intimately familiar with the entire instruction set allows you to generate the most efficient code.
[3] Recall that microcontrollers are generally designed to control other pieces of hardware. This means they must read individual "status" inputs and respond by sending out "control" outputs to the items the MCU is controlling.

1) **Understand the Problem:** This is an important step because if you don't understand the problem, there is no way you'll generate a viable solution[4]. The general idea here is that you'll get a high-level picture of the problem, which starts your brain thinking a path to the solution, which is of course the next item.

2) **Generate a Path to the Solution:** The notion of generating a path to the solution involves writing designing an algorithm that will solve the problem using the given parameters. In reality, there is no way you can solve the entire program with one giant plop: your brain does not work that way, and computer programming (as of this writing) does not work that way either. You'll be designing an algorithm, and there are two standard approaches to algorithm design:

   a. Pseudo Coding: Pseudo code is an unstructured semi-written-language approach to describing a path the solution. We'll not cover this approach in this text, but it truly is helpful and something that all programmers should know.

   b. Flowcharts: The flowchart provides a visual description of the basic flow of your program. Flowcharting describes program flow by using a few basic shapes. We dedicate the bulk of this chapter to flowcharting, so we'll opt not to say much here.

3) **Translate the Path to Assembly Language Code:** Once you've mapped out your algorithm, you must then translate the algorithm into the actual assembly language instructions that will implement that algorithm on a given processor. It goes without saying that for this step, you'll need to meet the two requirements of solving problems using assembly language.

A few comments regarding all these new rules and things:

- You can't step item #1; you need to understand the problem before you can solve it.
- You can skip step #2, but you should not. In addition, if your program is anything other than simple, you won't be skipping step #2 if you plan to actually generate a viable solution to the problem.
- You can't do step #3 if you don't have a solid understanding of the instruction set.

17.3 Structured Programming

The official definition of a simple program is one that can be decomposed into simple parts. A consequence of this definition is that if your program can't be composed into simple parts, it's a complex program. There are many concerning issues with complex programs including the fact that they suck. Let's face it; if you're not a disciplined programmer, you're going to be writing spaghetti code, and you're going to hate life as much as you boss or instructor hates you. Here are some more specific issues:

- They have a lower probability of working in all cases
- They are hard to understand
- They are hard to maintain
- They are hard to modify and/or reuse parts

[4] You'll generate a bunch of code, but it will generally be worthless. You may get lucky, but engineers don't rely on luck; only administrators rely on luck (and a wild show of waving hands) to solve problems.

The approach you want to take is to write “structured code”. The basis of writing structured code is to realize that any code you can possible write can be categorized into one of three “structures”: 1) the sequence construct, 2) the if-then-else, and, 3) the iterative construct. In other words, your code is either 1) doing something, 2) deciding where to go to do something, or, 3) doing something over and over again. These constructs will become easier to understand after you understand flowchart, which we cover in the next section.

Rember, if you’re truly writing structured programs, your code is going to be a series of these three constructs. In other words, you should be able to decompose your program into a set of these three structures. Disclaimer: just because you’re writing structure code does not guarantee that the program is going to work properly, as there are other issues present that you need to contend with. The payoff is that structured code is essentially the most cost effective approach to obtaining a working program. Structured programming has the added benefit of helping new programmers learn to work with the instruction set and develop their own great programming style.

17.4 Motivational Discussion of Flowcharting

You can view the writing of any useful software (or firmware) a solution to some problem. In other words, any worthwhile program that was ever written was done in order to do something useful. We can characterize this usefulness as providing a meaningful result. This being the case, we could further characterize the solution as an *algorithm*.

As you’ve probably read somewhere before, an algorithm is a computational or logical method of producing a desired result. A flowchart facilitates the development and visual representation of an algorithm, which is why they are so useful. The flowchart is the software analog to the block diagram used to describe hardware subsystems. Recall that hardware block diagrams are able to quickly convey an understanding of the circuit at hand. You’ll find that flowcharting an algorithm serves the same purpose: flowcharts quickly convey the basic operation of an algorithm. Keep in mind that both flowcharts and block diagrams worked well with hierarchical design to further promote understanding of the items they represent.

A flowchart has two basic purposes. It is the best idea to consider it a design tool, which is how we’ll be emphasizing it here. But being that flowcharts present a graphical representation of the order in which operations are carried out by programs, we can also consider them a great documentation tool that provides another description of your program in addition to the code (well commented code) itself. The use of flowcharts as a documentation tool is a by-product of proper program design. The reality is that the flowchart will be a great aid for anyone who needs to design a program. When the program is complete, the flowchart automatically becomes a great documentation item. In the end, the flowchart is great design tool and documentation tool.

We can judge any piece of program code by the following qualities (with lots of overlap among these qualities): modularity, reusability, understandability, readability, modifiability, and extendibility. If you can write code that contains all of these qualities, you win the big prize of having reliable code. It’s a big and sometimes ugly programming world out there. Quite possibly, many of the software designs you’ve completed were relatively small and with a single or close to being single purpose in mind.

In the real world, you’ll mostly likely be working on a team of people who all in one way or another is contributing to the production of a given piece of software. As you can imagine, it’s a big piece of software since there are so many people are working on it. In this case, if even one small part of the code does not contain all of the above qualities, the code will spawn many problems that have a strange tendency of never going away. Problems that don’t go away will create a lifetime of problems for anyone and everyone who has any dealings whatsoever with the project. The result is an unmaintainable, unmanageable, and worst of all, unreliable piece of software crap that people will continuously marvel at the fact that it ever works at all.

Flowcharting supports all the qualities of good source code. So if the discussion above has not convinced you that you should use flowcharts in your program design and subsequent documentation, just do it anyway. Someday you’ll thank yourself for building a sound foundation of solid programming practices.

The overall purpose of flowcharts is to quickly present information regarding a process or algorithm (particularly one code using a programming language). This is the overall goal but also keep these few items in mind when dealing with flowcharts.

- There are many options so as how to generate flowcharts, we'll stick with the basic symbols. You can add the bells and whistles later as you see fit.

- There is no "right" method to do flowcharts. In that they are tools to help you design and/or document your work, you'll need to provide your own definition of "right". A good place to start, however, is with the basic concepts presented in this chapter.

- If your flowchart meets the overall goal stated above, you have a good flowchart. Part of this definition of "right" should be the level of detail that your flowcharts provide. You many need to have several flowcharts for one section of code where each of the flowcharts would present data at a different level. Flowcharts do quite well presenting various levels of detail.

- Don't hesitate to present "flowcharts within flowcharts" because as you'll see, they nest quite nicely. The only rule you should follow is that any single flowchart should contain about the same level of detail (note the ambiguity of the word "about"). If you need to change that level of detail, you should start a new flowchart.

17.5 The Basics of Flowcharting

Table 17.1 shows the a few basic symbols that we typically use in flowcharts. When you see flowcharts in various places, you'll be seeing other symbols also, but these other symbols represent bells and whistles. As far as structured programming goes, the symbols in Table 17.1 represent the basic functionality of sequential programs, so the discussion in this chapter sticks with those symbols. We'll start continue this discussion by looking at the flowcharts as they relate to some of the basic programs we've written so far.

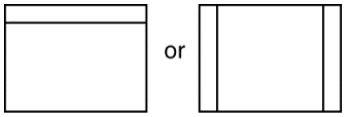
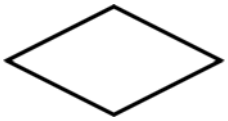
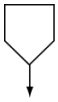
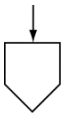
Symbol	Description
	Flow lines and flow arrows: the directed line segment indicates a sequence that the program follows. These lines guide the reader through the other flowcharting symbols in the correct order.
	Process: The rectangle symbol indicates that the algorithm performs the operation or process listed in the rectangle. All process symbols will have only one exit flow line but can have multiple entry points.
	Predefined Process: These are a special type of process symbols that are generally used to specify a process that is predefined (such as a subroutine) or defined in some other location.
	Decision: The algorithm determines program flow by the condition specified inside the diamond. The decision symbol will have not more than two exit flow lines, which are either yes, or *no*. Decision boxes and have multiple entry points.
	Terminal: specifies the beginning or end of a program or subroutine.
	Off-Page Connection, Entry: This symbol indicates that a given flow line continues on another page. These symbols are generally filled with a short label such as “A”, that match the off-page exit connection.
	Off-Page Connection, Exit: This symbol indicates that a given flow line continues on another page. These symbols are generally filled with short labels that match the off-page entry connection. .

Table 17.1: The basic symbols used in flowcharting.

The first and simple program we looked at was an infinite loop that didn’t do too much. Figure 17-1(a) shows this program while Figure 17-1(b) shows the associated flow chart. The fact that the program is simple makes it a good place to start describing flowcharts. Here are some of the more interesting facts regarding this program and the associated flowchart.

- Since Figure 17-1(a) shows the complete program, the code uses the terminal symbol labeled *start* to show where the important things in the program happen. Note that there is no ending terminal symbol since this program is essentially an endless loop.

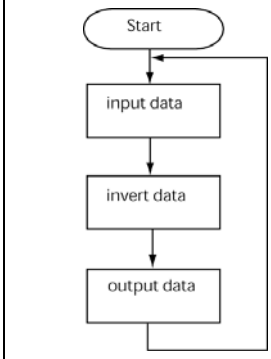
- The first instruction in the program is actually the CLI instruction; the flowchart has opted to omit this instruction. This is an acceptable choice with the justification that the instruction is not part of the algorithm.

- The flowchart is written at a relatively low level. Note that the flowchart includes practically every instruction from the associated program in a process symbol. This is arbitrary; an acceptable substitution for this flowchart would contain only one process symbol that contained the information included in the three process symbols listed in Figure 17-1(b).

- Although the other instructions each are contained in a process symbol, the BRN instruction is not. This is because the BRN instruction is more of a flow indicator rather than a request to do something, as would be the case for items placed in process symbols. Also note that this is an unconditional branch instruction; we handle conditional branch instructions differently.

- The arrow representing the flow control for the unconditional branch instruction feeds back to another flow line as opposed to a process box. We could have placed the arrow pointed at the top-most process symbol without losing the meaning of the flowchart. This is a preference of the person designing the flowchart, but the better choice will be to point flow line arrows only to other flow arrows. Whatever you decide, try to be consistent.

```
;-----------------------------------------------------------------
;- Assembler Directives
;-----------------------------------------------------------------
.EQU SWITCHES = 0x02          ; input port for switches
.EQU LEDS     = 0x0C          ; output port for LEDs
;
;-----------------------------------------------------------------
.CSEG                         ; indicates code segment
.ORG   0x00                   ; sets the code segment counter to 0x00
;-----------------------------------------------------------------

;-----------------------------------------------------------------
            CLI                     ; prevent interrupts
;
main:       IN      r0, SWITCHES    ; put switch status into register r0
            EXOR    r0, FF          ; complement switch data (trick)
            OUT     r0, LEDS        ; write data to LEDs
            BRN     main            ; main loop (do it again)
;-----------------------------------------------------------------
```



(a) (b)

Figure 17-1: Assembly code for the first program (a) and the associated flowchart (b).

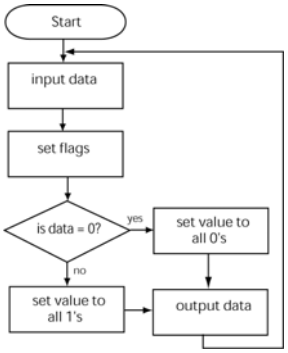
Figure 17-2 shows the assembly code for the second program and associated flowchart. Note that this program is a representation of an if/else structure and therefore contains a conditional branch instruction. The overall objective of this piece of code is to perform some operation based on a condition. Here is a list of worthy things regarding the assembly code and associated flowchart of Figure 17-2.

- The second instruction sets the condition flags, namely the Z flag. If the data input from the previous instruction is zero, the Z flag will be set; otherwise, the Z flag is not set.

- A decision symbol represents the conditional branch instruction. Program flow will continue in one of two directions based on the condition of the Z flag.

- Independent of the path taken because of the conditional branch instruction, the data will be output to the output port. The algorithm then repeats the entire process *ad nauseum*.

```
;-----------------------------------------------------------------------
;- Assembler Directives
;-----------------------------------------------------------------------
.EQU  DATA_IN_PORT       = 0x56   ; example input port data
.EQU  DATA_DATA_OUT_PORT = 0x57   ; example output port data
;
.CSEG
.ORG  0x00        ; memory location of instruction data
;-----------------------------------------------------------------------
; if/else construct
;-----------------------------------------------------------------------
;           CLI                     ; prevent interrupts

main:       IN      r0,DATA_IN_PORT   ; grab data
            TEST    r0                ; set condition flags
            BRNE    not_zero          ; jump if r0 is non-zero
            MOV     r1,0x00           ; clear bits in s1
            BRN     out_val           ; jump to output instruction
                                      ;
not_zero:   MOV     r1,0xFF           ; set bits in s1
out_val:    OUT     r1,DATA_OUT_PORT  ; output some data
            BRN     main              ; do it again
;-----------------------------------------------------------------------
```



(a) (b)

Figure 17-2: Assembly code for the second program (a) and the associated flowchart (b).

Figure 17-3 shows the assembly code for the third program and associated flowchart. Note that this program is a representation of an iterative loop construct, which is one of the most useful constructs in any programming language. This set of code also contains a conditional branch instruction but is used somewhat differently from the previous program. The overall objective of this piece of code is to perform some operation a set number of times. The number of times is loaded into the s3 register prior to entering the main body of the loop. Here are some other things of merit regarding the assembly code and associated flowchart of Figure 17-3.

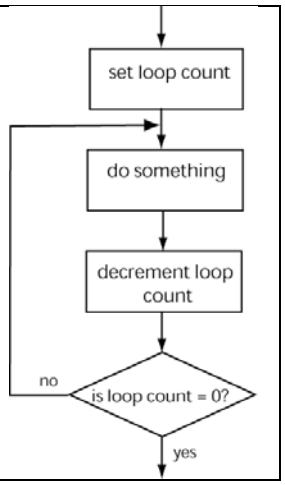
- The code in Figure 17-3(a) is not a complete program. We often refer to snippets of code such as this as program fragments in computerland and we use this terminology throughout this discussion.

- Before the program enters the loop, we first initialize the loop counter to the number of times the program needs to execute the loop.

- The body of the loop does some pointless task; the code you write will have more meaningful tasks than the task in Figure 17-3(a).

- The programmer must do two things within every loop body: 1) decrement to loop count, and 2) check the value of the loop count and see if it is zero or not. If the loop count is zero, exit the loop (don't execute it again), otherwise the loop is executed again. The decrement instruction in Figure 17-3(a) updates the state of the zero flag; the program then tests the zero flag in the conditional branch instruction that follows.

```
;-----------------------------------------------------------------------
;- Assembler Directives
;-----------------------------------------------------------------------
.EQU  DATA_IN_PORT       = 0x56   ; example input port data
.EQU  DATA DATA_OUT_PORT = 0x57   ; example output port data
;
.CSEG
.ORG  0x00                        ; memory location of instructions
;-----------------------------------------------------------------------
; iterative loop construct
;-----------------------------------------------------------------------
;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
            MOV     r3,0x08             ; load iterative count value
                                        ;
loop:       IN      r1,DATA_IN_PORT     ; grab data
            EXOR    r1,0xFF             ; toggle input data
            OUT     r1,DATA_OUT_PORT    ; output some data
            SUB     r3,0x01             ; decrement iteration variable
            BRNE    loop                ; do it again if count non-zero
                                        ;
loop_done:  ; do something else…

;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;-----------------------------------------------------------------------
```

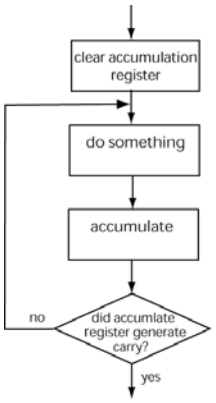


(a) (b)

Figure 17-3: Assembly code for the third program (a) and the associated flowchart (b).

Figure 17-4 shows the assembly code for the fourth program and associated flowchart. Note that this program is a representation of a conditional loop construct. The main difference between this code and the iterative loop code presented in the previous program is the fact that the program does not know in advance how many times it will execute the body of the conditional loop. This is because the number of iterations depends upon the data input in the body of the loop. Other than this fact, this code is very similar to the code for the iterative loop.

```
;-----------------------------------------------------------------------
;- Assembler Directives
;-----------------------------------------------------------------------
.EQU  DATA_IN_PORT       = 0x56   ; example input port data
.EQU  DATA DATA_OUT_PORT = 0x57   ; example output port data
;
.CSEG
.ORG  0x00                        ; memory location of instructions
;-----------------------------------------------------------------------
; conditional loop construct
;-----------------------------------------------------------------------
;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
            MOV     r0,0x00             ; clear register
loop:       IN      r1,DATA_IN_PORT     ; grab some data
            EXOR    r1,0xFF             ; compliment bits
            OUT     r1,DATA_OUT_PORT    ; output some data
            ADD     r0,r1               ; add value to accumulation r0
            BRCC    loop                ; repeat is no carry generated
            ;
loop_done:  ; do something else…
;
;~~~~~~~~ program fragment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;-----------------------------------------------------------------------
```



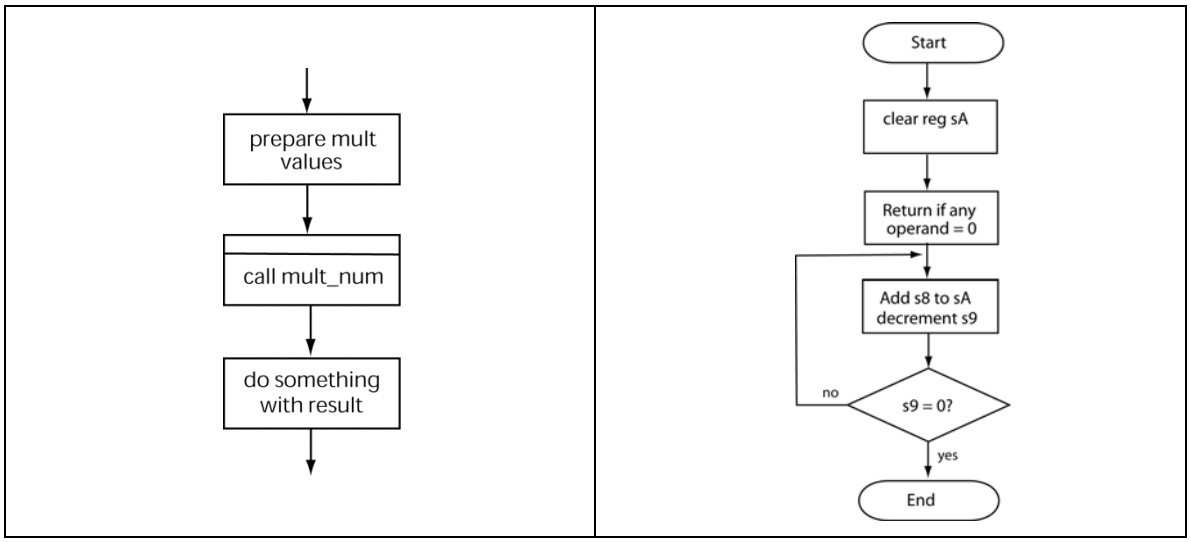
(a) (b)

Figure 17-4: Assembly code for the fourth program (a) and the associated flowchart (b).

And for our last example, let's create a flowchart for the four-bit multiplier subroutine and also a flowchart that calls that code. Figure 17-5 shows the code for the mult_num subroutine and example calling code. Figure 17-6 shows a flowchart for both the calling code and the actual subroutine itself.

```
;-------------------------------------------------------------
; sample calling code fragment for mult_num subroutine.
;-------------------------------------------------------------
            MOV    r8,0x05     ; prepare registers r8 & r9
            MOV    r9,0x0B     ;  to do multiply
            CALL   mult_num    ; do the multiply
            MOV    r0, r10     ; use the result

;-------------------------------------------------------------------
;- subroutine: mult_num  - places the effective multiplication
;-  of the number in register r8 with the number in register r9. It
;-  is expected that these number are limited to 4-bits or the
;-  result in register r10 will not be valid.
;-
;- Affected Registers: r9,r10
;-------------------------------------------------------------------
mult_num:   MOV     r10,0x00    ; clear r10 to act as accumulator
            CMP     r8,0x00     ; set flags
            BRNE    ret1        ; jump to check other operand
            RET                 ; return 0 if operand is 0

ret1:       CMP     r9,0x00     ; see if other operand is zero
            BRNE    loop        ; jump to do calculation
            RET                 ; return if it is 0
                                ;
loop:       ADD     r1,r8       ; add multiplicand
            SUB     r9,0x01     ; decrement other operand
            BRNE    loop        ; jump if r9 is not zero (add again)
            RET                 ; r9 is zero; result is in r10.
                                ; return control to the main program
;-------------------------------------------------------------------
```

Figure 17-5: Source code for a 4-bit multiplier subroutine.



(a) (b)

Figure 17-6: The flowcharts for the calling code (a) and *mult_num* subroutine (b).

Figure 17-6(a) shows the flowchart associated with the code that calls the subroutine. Note that the flowchart uses a simple process box in order to indicate the subroutine call. You also can list the subroutine flowchart itself in another document or on another page. For our case, Figure 17-6(b) lists the flowchart for the subroutine. Here are a few other items to note about this solution.

- The flowchart for the subroutine has both a *start* and *end* terminal box. This is in stark contrast to the main code we've looked at which only had a start terminal symbol.

- In the flowchart for the subroutine, some of the process boxes have covered the equivalent of several assembly instructions. This is an example of a higher-level process boxes, which we've not really used in the previous examples.

17.6 Structured Programming Revisited

We gave a motivation blurb regarding structured programming in an earlier section; we now need to fill in the details. Recall that the notion of structured programming is that any well written program can be decomposed in to a conglomeration of three basic structures: 1) the sequence structure, the if-then-else structure, and 3) the iterative structure. As you'll see, we use two or more of the basic flowcharting main symbols to model each of these structures: the process box, the decision box, and associated flow lines. Not surprisingly, flowcharts are probably the best way to define/understand these three basic structures.

17.6.2 The *sequence* Structure

The sequence structure is a set of two or process boxes placed in a series and considered as a new "higher-level" process box. Figure 17-7(a) shows the basic model of a sequence structure using standard flowcharting symbols. The notion of a sequence should seem familiar, as it is simply a form of abstracting to a higher level. The main characteristic of a sequence structure is that it begins at one point and ends at another, which is simply a way of stating that the sequence structure has on entry point and one exit point. If a structure has more than one entry point or more than one exit point, then it is not a sequence structure and necessarily a part of structured programming. In this case, you can possibly model it as a sequence structure if you further decompose the objects such that you can model them using standard structures.

17.6.3 The *if-then-else* Structure

The if-then-lese structure represents a decision point: the program decides to take one path or another based on some condition in the program. Figure 17-7(b) shows the basic model of an if-then-else structure using standard flowcharting symbols. To be a true if-then-else structure, the two paths must eventually merge after the execution of the chosen path completes. This characteristic assures that the if-then-else structure is similar to the sequence structure in that the if-them-else structure has one entry point and one exit point. Keep in mind that a variation of the if-then-else structure is the if-then structure. In this case, the structure either does something or does nothing, as compared to the if-then-else structure that does something or does something else.

A specialized form of the if-then-else structure is the in-case-of structure. This is similar to the notion in higher-level languages of using if-then-else statements or case statements to implement the same functionality.

17.6.4 The *iterative* Structure

The iterative structure models a set of instructions that repeatedly performs the same process until the structure makes the decision to exit the structure. Figure 17-7(c) shows the basic model of a iterative structure using standard flowcharting symbols. We use the notion of an iterative structure to model both types of iterative

constructs: the while-loop and the do-while loop[5]. The iterative structure is independent of the terminating condition, meaning that the terminating condition can be any condition supported by the exact form of the underlying language's flow control statement[6]. Similar to the sequence and if-then-else construct, the iterative construct has one entry point and one exit point.

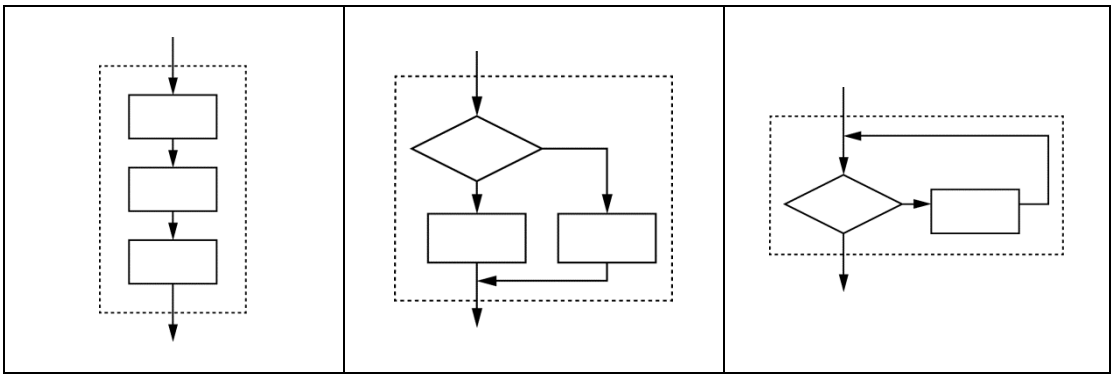


Figure 17-7: Basic models for (a) sequence structure, (b) an if-then-else structure, and c) an iterative structure.

17.7 The Truth about Software

Software is mysterious. Have you ever seen a program running? There's a program running the machine that's keeping your grandmother alive during her hospital stay. Do you know who wrote that program? Do you know how extensively they tested that program? Did the person in charge of that software think of every possible scenario before they released the code? Should you be worried about all this stuff? I'm not sure what the answer is, but if you were worried about whether all the software that runs the world is really working correctly, you'd probably need to take lots of medication to make it through the day.

If you're reading this sentence, you're probably embarking on learning to write assembly language. Yep, it's real fun to make the LED blink or the numbers count; it sure seems trivial. However, it's really rather important. Someday you'll graduate and be tossed onto a team developing a new product. You'll be surprised how instantly that team starts depending on you two write for their next product; it will sort of wish you started writing good code from the get-go. When you see your company's product on the shelf or flying through the air, are you going to be wondering if your code really works or not?

17.7.1 Software Quality

Does your software work properly? How would you know if you did not extensively test it? Do you think your boss is going to ask you if you extensively tested your software? No, he or she is not. What they're going to be asking you is if your software is completed or not. If you ask for more time to test it properly, you won't be considered a team player and probably be soon laid off.

In reality, the testing and debugging of your software is most likely going to require more time than it required you to get to that point (which includes planning and writing the software). In most jobs, you'll barely have

[5] Recall that the do-while loop always executes the associated process at least one time, which is does by executing the process before it checks the terminating condition. The while-loop checks the terminating condition before executing the assicated process and thus can exit the loop before executing the process.
[6] In particular, we're referring to the conditional branch instructions in the underlying assembly language.

enough time to design and write the software before the release date; testing is not usually a high priority. Sad to say that the only thing that has a lower priority than testing is documentation.

Your mission is still to write good code. Good code is going to work, and if it does not work, it's going to be easy to debug. Code that is easy to debug is presents a shorter path to getting the code to work. But let's be real here: all the testing in the world won't guarantee that your code will work 100% of the time. What testing will show you is that your code has bugs; testing is not going to show you that your code does not have bugs. All is not lost here; there are a few simple rules to follow to help you write good code. If you're conscientiously striving to write good code, your code will be in a constant state of improvement. If you learn from your mistakes, you won't be making those mistakes again.

17.8 Writing Good Programs

There are many great books out there describing various techniques you can use to write good programs. Because you are probably a student in an academic environment, you generally don't get a chance to experience the normal "real world" approach and accompanying expectations of writing real software. In academia, the main goal of your software is to complete the assignment. In this case, you know full well that your program is probably being graded by a robot, which means most of the corners you cut attempting to submit the assignment before the due date will go unnoticed by any other human.

There are several problems with writing code in an academic environment. First, courses in academia typically place way too much emphasis on completing the assignment ***at any cost***. You programs does what it should in that it made the robot grader happy, but at what cost? Your code may be crappy, unreadable, unorganized, unmaintainable code—the robot does not care. Because no human outside of yourself ever sees the code to inform you of your diminished code quality, you develop bad habits that you may never break. In addition, in academia, you generally have the choice of obtaining any grade outside of an F and still attain success on the assignment and pass the class. If truth, if you apply the same approach outside of academia[7], you'd be fired rather quickly.

There is a right way to write code. Though you may not always have the time to take this approach, you know you should be taking the approach. We all strive to be lucky enough to have the time and/or resources to embark on writing good programs. There is much more to writing good programs than plopping down some instructions. The final word here is that writing good code is a process that extends well beyond regurgitating instructions and/or expressions; enjoy the journey.

1) **Know how to program:** There is more to programming than simply writing code. Anyone can write good code, but it's truly a learning process. The main problem in academia is that lazy professors don't take the time to ensure their students are writing good code. The typical lazy professor typically verifies the code appears to be working (or has a robot check to see if it's working) and quickly moves on to check-off the next program.

    a) Make your code look good: Good-looking code is code that looks good standing ten feet back. In truth, most people (non-robots) who look at your code are only going to take a cursory glance it; people rarely take the time to determine if your code is actually good or not. Just like most everything else in life, people will make a snap judgement based primarily on appearances: if you code looks good, it is good. So if you for some reason are not writing good code, the least you can do is make it look good.

    b) Strive to write structured programs: Structured programs are easier to "get working", understand, maintain, and most importantly, debug. To be able to write structured programs, you must understand the three basic structured programming constructs: 1) sequences, 2) if-then-else, and 3) iterative constructs.

[7] Keep in mind this level of incompetence ensures you a promotion if you're an academic administrator.

c) Know the entire instruction set: If you don't know the instruction set, you'll never be able to write an assembly language program. If you writing in a higher-level language, knowing the underlying instruction set is going to help you write "better" code[8].

d) Know the tools: There are various tools that help you write good code in an efficient manner. Assemblers and compilers have a various options to help you write code that it more understandable and more portable[9]. Simulators/debuggers often have not overly apparent feature to help you ensure your code is working properly.

2) **Write simple code:** Simple code is has many things going for it, though job security is not one of them. Good programmers know and understand the notion that there is a certain eloquence and beauty to good code; it's a characteristic that defies description. If you're trying to impress people with your code, strive to impress them with the simplicity of your code. You may not impress your butthead friends and colleagues with your code, but other good programmers will be.

a) Write understandable code: the assembler does not care what your program looks like, but other humans do. Understandable code is easier to get working properly, including the eventual debug part of the process. If you pass crappy code along to colleagues, they'll quickly lose respect for you programming abilities. Be sure to find an approved style file and make you code look like the code in the style file (or preferably, better).

b) Comment your code: Use comments to primarily state "why" you're code is doing something is generally more important than stating "what" your code is doing. Avoid commenting on things are obvious. Keep comments brief, but be sure to add extra comments for code that is doing something strange of patently unobvious.

c) Use labels in your code: Labels cost nothing but do provide a vehicle for making your code more understandable. Recall that labels are generally short mnemonics that quickly transfer information.

d) Use white space: In fact, use a liberal amount of white space. Everything, including comments, directives, and instructions should be properly and consistently indented. Use blank lines to delineate separate ideas in the code stream. Also, use blank lines to delineate subroutines.

e) Write modular code: Possibly the main attribute of simple code is that it is modular. Modular code is easier to write, understand, reuse, debug, and maintain. The main vehicle for modules in assembly language programming is subroutines. Each subroutine should have a header that describes the purpose of the module, and what resources the subroutine changes.

f) Don't write tricky code: Well, sometimes you have to in the name of efficiency… But if you do write tricky code, make sure you comment the code with an excruciating amount of detail.

g) Write portable code: The notion of portable code means that if something in the underlying hardware changes (either the MCU of external hardware controlled by the MCU); your code will require little or no modification in order to work properly. Try not to write code that requires intimate knowledge of the hardware, or keep such knowledge to a minimum (and well-commented). Use directives defined in the initial portion of your code to define constants used by the hardware.

h) Use LUTs when possible: You can't say enough good things about LUTs. Always be on lookout for instances in your code where a LUT is appropriate (makes your code clearer and/or more efficient).

[8] But if you're writing in a higher-level language, often times you're doing it for its portability characteristics, so you may not know what anything about the underlying hardware.
[9] Recall that assembler directives are messages from the programmer to the assembler.

17.9 Chapter Summary

- Writing programs to solve problems is an art form. However, those learning the art can get a good start by not losing sight of the problem being solved and by following this simple set of guidelines.
- Structured programming using basic constructs assembled in a workable manner to write programs. Programs that are not properly structured often end up becoming “spaghetti code”, and are essentially, giant pieces of crap.
- Flowcharts provide a simple approach to program design and program documentation.
- Flowcharts as a design tool give programmers a visual representation of program flow, which is important in assembly languages as they can quickly become long and complicated.
- Flowcharts as a documentation aid will help others quickly understand the intended purpose and flow of your assembly language source code.
- Flowcharting is based on a few simple symbols including program flow, process, predefined process, decision, and terminal.
- The three basic structured programming structures are the sequence, if-then-else, and iterative constructs. If-then-else constructs include case-type constructs while iterative constructs include both do-while and while constructs.
- Your software is going to have bugs; the best you can hope for is to keep the number of bugs and the damage the bugs cause to a minimum.
- Verification and debugging of programs usually takes longer than the actual planning and writing of programs.
- Writing good programs is an art form. If you’re not an artist, you can follow a basic set of guidelines to prevent your code from becoming crappy.

17.10 Chapter Exercises

1) In your own words, describe the main purpose of an algorithm.

2) What is the hardware analog to a firmware flowchart?

3) What are the two main purposes of flowcharts?

4) Provide a flowchart for the following assembly language source code.

```
;-------------------------------------------------------------
;- subroutine: add_bcd
;-
;- This subroutine adds two 2-digit BCD numbers residing in
;- r10 & r11. The upper and lower nibble represent the 10's
;- 1's digit, respectively. The result is placed in r20 for
;- the 100's digit (lower nibble) and r21 (upper nibble for
;- the 10's digit and lower nibble for the 1's digit). This
;- subroutine does not check for valid BCD numbers in r10
;- & r11.
;-
;- Registers tweaked: r20, r21, r28, r29, r30, r31
;-------------------------------------------------------------
.EQU LO_NIB_MASK    = 0x0F
.EQU HI_NIB_MASK    = 0xF0
.EQU TENS_INC_VAL   = 0x10
.EQU LO_TEN         = 0x0A
.EQU HI_TEN         = 0xA0

add_bcd:                            ;
            MOV     r28,r10         ; copy input values
            MOV     r29,r11         ;
            MOV     r30,r10         ; copy input values
            MOV     r31,r11         ;

            AND     r28,LO_NIB_MASK  ; massage the input data
            AND     r29,LO_NIB_MASK
            AND     r30,HI_NIB_MASK
            AND     r31,HI_NIB_MASK

            ADD     r28,r29         ; do 1's digit add
            CMP     r28,LO_TEN      ; see if it exceeded 10
            BRCS    lt_10x          ; branch if it did not
            ADD     r30,TENS_INC_VAL ; increment 10's data
            SUB     r28,LO_TEN      ; subtract 10 from 1's digit

lt_10x:     ADD     r30,r31         ; add the 10's digit
            CMP     r30,HI_TEN      ; see if it exceeded 10
            BRCS    lt_10y          ; branch if it did not
            MOV     r20,0x01        ; increment the 100's spot
            SUB     r30,HI_TEN      ; subtract a up nibble 10
            BRN     done

lt_10y:     MOV     r20,0x00        ; clear the 100's nibble
done:       MOV     r21,r28         ; pack the upper and lower
            OR      r21,r30         ;  nibble
            RET
; -------------------------------------------------------------
```

5) Provide a flowchart for the following assembly language source code.

```
main:    MOV   r5,0x48    ; test driver
         CALL  bcd2bin
         BRN   main

;-------------------------------------------------------------
;- subroutine: bcd2bin
;-
;- This subroutine converts a two 2-digit BCD numbers residing in
;- r5 to an unsigned binary number. The upper and lower nibble
;- in r5 represent the 10's and 1's digit, respectively. The
;- resulting unsigned binary number is placed in r8; this
```

```
;- subroutine does not change the value of r5.
;-
;- Registers tweaked: r0, r8
;-------------------------------------------------------------
.EQU LO_NIB_MASK    = 0x0F
.EQU HI_NIB_MASK    = 0xF0
.EQU TENS_INC_VAL   = 0x10
.EQU LO_TEN         = 0x0A
.EQU HI_TEN         = 0xA0

bcd2bin:                            ;
            MOV     r0,r5           ; copy data
            MOV     r8,r5           ; copy data
            AND     r8,LO_NIB_MASK  ; accumulate lower nibble

            LSR     r0              ; move top nibble to low
            LSR     r0              ;  nibble
            LSR     r0
            LSR     r0
            AND     r0,LO_NIB_MASK  ; clear top nibble

loop:       CMP     r0,0x00         ; keep adding 10 for each
            BREQ    done            ;  value in 10's digit
            ADD     r8,LO_TEN       ; accumulate 10
            SUB     r0,0x01         ; decrement 10's count
            BRN     loop            ; keep doing stuff

done:       RET
; -------------------------------------------------------------
```

6) Provide a flowchart for the following assembly language source code.

```
;-------------------------------------------------------------------
;- Program: bounce
;-
;- This programs makes it appear as if one lit LED is bouncing
;- back and forth between the eight LEDs on the development
;- board. This program calls an imaginary subroutine that slows
;- the bouncing rate down to something humans can see and
;- comprehend, though the concept may still be too complicated
;- for academic administrators to comprehend.
;-
;-------------------------------------------------------------------
.EQU LED_PORT = 0x20             ; port for LED output --- OUTOUT
;-------------------------------------------------------------------
.CSEG
.ORG  0x10

init:    MOV   r8,0x01          ; init the one lit LED

main:    MOV   r7,0x07          ; load loop count
loop1:   OUT   r8,LED_PORT      ; output LED data
         CALL  Delay            ; wait
         LSL   r8               ; shift left
         SUB   r7,0x01          ; decrement loop count
         BRNE  loop1            ; continue if need be

         MOV   r7,0x07          ; load loop count
loop2:   OUT   r8,LED_PORT      ; output LED data
         CALL  Delay            ; wait
         LSR   r8               ; shift right
         SUB   r7,0x01          ; decrement loop count
         BRNE  loop2            ; loop if necessary

         BRN   main             ; lather, rinse, repeat
;-------------------------------------------------------------

;-------------------------------------------------------------
;- Subroutine: Delay
;-
;- This is a stub; it does nothing useful, similar to an
;- academic administrator.
;-------------------------------------------------------------
Delay:  AND   r0,r0
        RET
; -------------------------------------------------------------
```

7) Provide a flowchart for the following assembly language source code.

```
;-------------------------------------------------------------------
;- Program: Interrupt Delay Thang
;-
;- This program constantly reads from port 0x22, complementing
;- the data, and writing the data to port 0x23. The background
;- task implements a 2-interrupt delay. When the RAT MCU receives
;- an interrupt, it reads a value from port 0x55. The value that
;- was read from port 0x55 two interrupts earlier is then output to
;- port 0x57. The program outputs 0xFF for the first two writes in
;- the background task.
;-------------------------------------------------------------------
.CSEG
.ORG  0x10

init:    MOV   r20,0xFF         ; init ISR delay registers
         MOV   r21,0xFF         ;
         SEI                    ; allow interrupts

main:    IN    r0,0x22          ; get some data
         EXOR  r0,0xFF          ; complement data
         OUT   r8,0x23          ; output LED data
         BRN   main             ; repetition is a good thing
;-------------------------------------------------------------

;-------------------------------------------------------------
;- Subroutine: ISR
;-
;- This subroutine implements the interrupt service routine.
;- This ISR implements a two interrupt delay. During this ISR,
;- a value is read from an input port; the value read two
;- interrupts earlier is output. The first two output values
;- output are 0xFF.
;-------------------------------------------------------------
ISR:
        OUT    r20,0x57         ; output oldest data
        MOV    r20,r21          ; transfer data to create delay
        MOV    r21,r22
        IN     r22,0x55         ; get new data
        RETIE                   ; go back to foreground task
;-------------------------------------------------------------

;-------------------------------------------------------------
.CSEG
.ORG  0x3FF
        BRN   ISR:
; -------------------------------------------------------------
```

8) Provide a flowchart for the following assembly language source code.

```
;-------------------------------------------------------------------
;- Program: Simple Averaging Filter
;-
;- This RAT interrupt driven program implements a digital filter.
;- The main code counts the number of interrupts received and
;- outputs this binary value to port 0x88 each time the RAT
;- receives an interrupt. This count is an 8-bit binary and rolls
;- over to zero when the count exceeds 0xCC. When the RAT receives
;- an interrupt, it reads a value from port 0x98; this value is
;- averaged with the value read during the previous interrupt
;- and the result is written to port 0x99.
;-------------------------------------------------------------------
.CSEG
.ORG  0x10

init:    MOV   r20,0x00         ; init ISR register
         MOV   r16,0x00         ; clear flag ISR flag register
         MOV   r10,0x00         ; clear ISR counter register

         OUT   r10,0x88         ; output ISR count value
```

```
        SEI                 ; allow interrupts

main:   TEST  r16,0x01      ; see if interrupt flag set
        BREQ  main

        ADD   r10,0x01      ; increment ISR count value
        CMP   r10,0xCC      ; max count exceeded?
        BRCC  no_rst
        MOV   r10,0x00      ; reset ISR count

no_rst: OUT   r10,0x88
        SEI
        BRN   main          ; repetition is read good
;-------------------------------------------------------------

;-------------------------------------------------------------
;- Subroutine: ISR
;-
;- This subroutine implements the interrupt service routine.
;- This ISR implements an averaging filter which averages
;- the two more recent values read from during the ISRs. The
;- calculation includes a rounding up feature.
;-------------------------------------------------------------
ISR:
       MOV    r20,r21       ; save more recent data
       IN     r21,0x98      ; get new data
       MOV    r30,r20       ; copy old data
       ADD    r30,r21       ; add new data
       LSR    r30           ; divide by 2
       ADDC   r30,0x00      ; add the carry for roundup
       OUT    r30,0x99      ; output the average

       MOV    r16,0x01      ; set an interrupt flag
       RETID                ; go back to foreground task
;-------------------------------------------------------------
;-------------------------------------------------------------
.CSEG
.ORG  0x3FF
       BRN   ISR
; -------------------------------------------------------------
```

9) Provide a flowchart for the following assembly language source code.

```
.CSEG
.ORG  0x10

main:   MOV   r20,0x64    ; test driver
        CALL  Bin2bcd
        BRN   main
;-------------------------------------------------------------------
;- subroutine: Bin2bcd
;-
;- This subroutine converts an 8-bit unsigned binary number
;- into a 3-digit BCD value. The unsigned binary number is
;- provided in r20 and the resultant BCD number in r25 & r26,
;- where the lower-nibble of r26 holds the 100's digit, the
;- upper-nibble of r25 holds the 10's digit, and the lower
;- nibble of r25 holds the 1's digit. This subroutine does
;- not alter the value in r20.
;-
;- Registers tweaked: r25, r26
;-------------------------------------------------------------------
.EQU CIEN     = 0x64
.EQU TEN      = 0x0A

Bin2bcd:
            PUSH    r20             ; save the value
            MOV     r25,0x00        ; clear result register
            MOV     r26,0x00

hunds:      CMP     r20,CIEN        ; compare with 100
            BRCS    tens            ; go onwards if no 100's
            ADD     r26,0x01        ; increment 100's count
            SUB     r20,CIEN        ; adjust original value
            BRN     hunds           ; keep doing it

tens:       CMP     r20,TEN         ; compare with 10
```

```
            BRCS    ones            ; go onwards if no 10's
            ADD     r25,0x01        ; increment 10's count
            SUB     r20,TEN         ; adjust original value
            BRN     tens            ; keep doing it

ones:       LSL     r25             ; shift lower nib to up nib
            LSL     r25
            LSL     r25
            LSL     r25
            AND     r25,0xF0        ; clear bottom nibble

done:       OR      r25,r20         ; add the 1's sloppy seconds
            POP     r20             ; restore the original data
            RET                     ; go back, all the way back
; -------------------------------------------------------------------
```

PART FOUR: RAT MCU Architectural Details

18 RAT Architecture Details

18.1 Introduction

Many of the previous chapters dealt the RAT MCU at a programmer's level. We took this approach as we were protecting the programmer's sanity by shielding them from the RAT MCU's lower-level hardware details. This chapter delves into those details by describing the underlying hardware details of the RAT MCU's submodules at both local and system levels. In other words, when you execute a RAT MCU instruction, there are certain actions the RAT MCU's hardware must take to correctly implement that instruction. This chapter describes the various submodules and their relation to the RAT MCU's instruction set.

Main Chapter Topics

- **DESCRIPTION OF RAT MCU'S SUBMODULES:** This chapter describes the various submodules in the RAT MCU architecture. These sub modules include the control unit, the program counter, the program memory, the scratch RAM, the Input/Output, subroutines, and the interrupt architecture.
- **PROGRAM FLOW CONTROL:** This chapter describes the various flow control mechanisms in the RAT MCU including branches, subroutines, and the RAT MCU interrupt architecture.
- **OVERVIEW OF RISC AND CISC ARCHITECTURES:** This chapter describes the RISC and CISC architectures including their main accepted differences.
- **OVERVIEW OF LEVELS OF MEMORY:** This chapter describes the notion of memory levels as they relate to basic computer systems.

Why This Chapter is Important

This chapter is important because it describes the low-level architecture details of the RAT MCU on both a local and system-level.

18.2 The Control Unit

The RAT MCU divides the execution of an instruction into two distinct steps. In essence, the execution of a program involves the repeated processing of these two steps. We officially refer to these two steps as the *instruction fetch cycle* and the *instruction execute cycle*, or the fetch cycle and the execute cycle. The instruction fetch cycle roughly involves retrieving the instruction contained in a program memory location while the execution cycle involves completing the operations associated with a particular instruction.

The RAT MCU's control unit controls the basic operation of the computer. The control unit is nothing more than a finite state machine (FSM), recalling that we consider FSMs as hardware entities that control other hardware

entities. We can nicely control the basic operation of the RAT MCU with a two-state FSM, where the two states are the "fetch" and "execute" states. Figure 18-1 shows the state diagram that controls the overall operations of the RAT MCU[1].

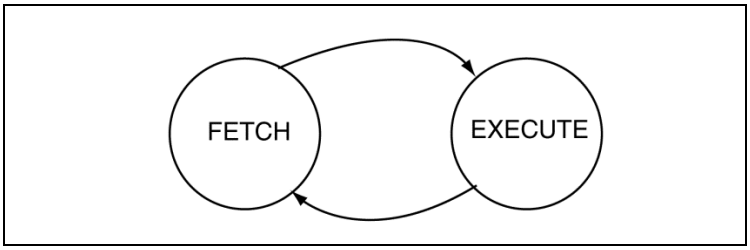


Figure 18-1: A state diagram that models the basic operation of the RAT MCU's control unit.

Figure 18-2 shows how the system clock delineates the fetch and execute cycles (assuming that the states change on a rising clock edge). Remember, all instructions in the RAT MCU's instruction set each require two clock cycles to execute: the fetch and execute cycle. The execution of one entire instruction requires two clock cycles; once again, we refer to these two clock cycles (the fetch and execute cycles) as an instruction cycle. Figure 18-2 shows two and one half instructions cycles. Typical computer architectures divide instruction execution into various cycles, sometime called "T-cycles", the RAT MCU has two T-cycles (fetch and execute); other computer architectures can have as few as one T-cycle or many more[2].

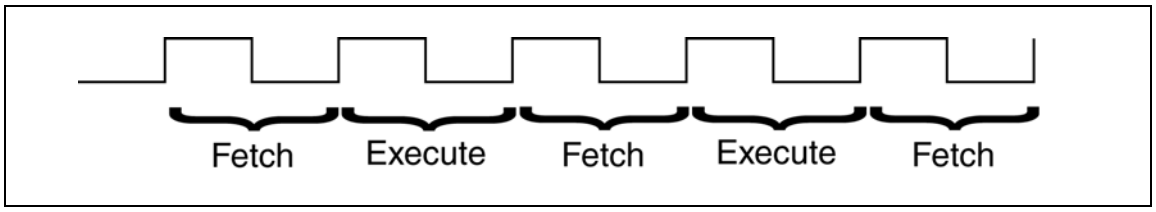


Figure 18-2: The system clock is divides the instruction cycle into fetch and execute cycles.

We typically think of FSMs as hardware that controls other hardware in a structured manner. The FSM for the RAT MCU, or the control unit, is no different. Figure 18-3 show the high-level block diagram of the RAT MCU's control unit. Generally speaking, the signals entering the black box on the left are status signals (not including the CLK signal) while the signals on the right are control signals.

In high-level terms, the control unit "fetches" the instruction opcodes during the fetch cycle and completes implementation of the instruction by sending out the appropriate control signals based on the fetched opcodes during the execution cycle. Note that in Figure 18-3 we group the outputs based on the basic functional submodules of the RAT MCU. In the order of the left-side outputs of the RAT control unit, these submodules include interrupt hardware, program counter, stack, register file, ALU, scratch RAM, and condition flag control. We cover the issues regarding the submodules in later sections of this chapter, so don't worry if they seem strange now.

[1] This state diagram is actually not complete, but it's good enough for now; we'll fill in the details later.
[2] Unlike many people, I'm not pretending to know all about computer architectures here; I have seen up to five cycles on one computer architecture. There certainly can be more.

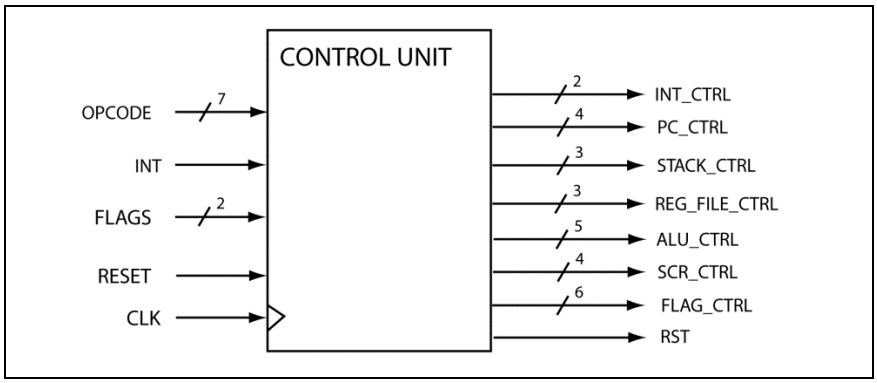


Figure 18-3: High-level black box diagram of the RAT MCU's control unit.

Figure 18-4 shows a relatively high-level diagram of the more interesting timing signals associated with the control unit. This diagram shows two full instruction cycles. Here are the important issues you should understand regarding Figure 18-4:

- The diagram uses the term "data" quite often. The data in these fields is not necessarily the same, but it could be in some cases. We use the term "data" to alert the reader to the notion that the lines contain data and this data is changing as the diagram indicates (using the "X" notation on the bundled signals).

- The CLK signal is the system clock signal that synchronizes all RAT MCU operations including the control unit. The RAT MCU uses the CLK's rising edge as the active edge.

- The OPCODE signals update after the rising-edge that marks the start of the fetch cycle. The data on the OPCODE line is seven of the 18 bits of data from the program memory. The data on the OPCODE signals represents the signals associated with the instruction pointed at in program memory at the beginning of the fetch cycle.

- The CTRL signals represent the signals the control unit outputs because of decoding the OPCODE input signals. These signals are different for different instructions with the goal of controlling the submodules required to implement the decoded instruction. Note that the control unit does not send out these signal until the start of the execute cycle, which is one clock cycle after the opcodes signals are available. The control unit is a FSM so it must change from the fetch state to the execute state in order for the control signals to become available to the various modules that use them.

- One clock cycle after the control signals are available, the status signal inputs to the control unit become available; this time also starts the fetch cycle. Part of the control unit's responsibility is sending out control signals to the condition flags such that they load the values from the condition signal resulting from the execution of some instructions (instructions that change the values in the condition flags). The control unit knows to send these signals after it decodes the current instruction's opcode.

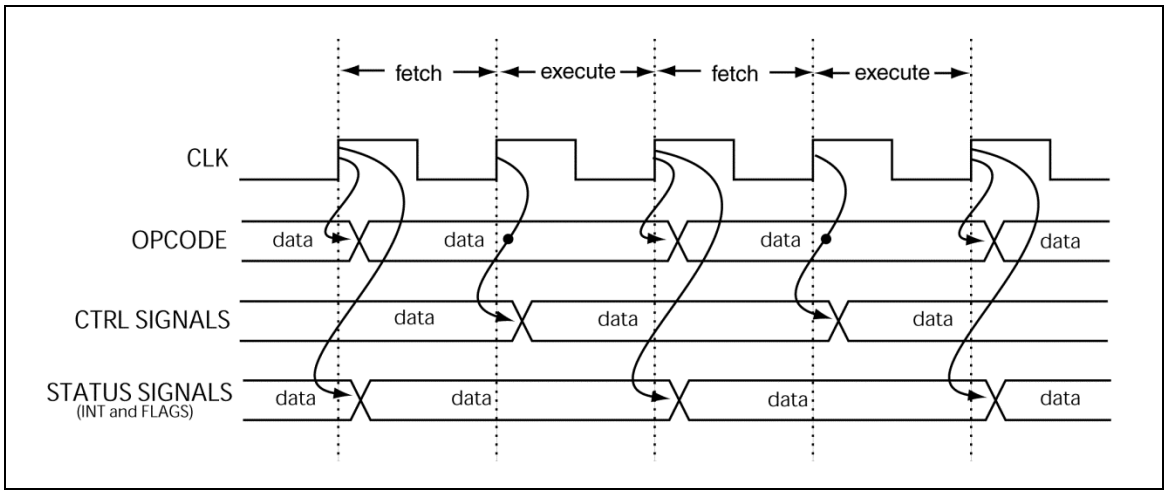


Figure 18-4: High-level timing diagram for the RAT MCU's control unit.

18.3 The Program Counter and Program Memory

The program counter, or "PC", is probably the most common sub-module in computer architecture. The PC's basic responsibility in a computer architecture is providing a pointer (address) to an instruction in program memory. The main idea behind a PC is that it generally points to the instruction in program memory that the MCU is currently executing. As you'll soon see, depending on the particular clock cycle, the PC points either to the currently executing instruction or to the instruction following the instruction that is currently executing. As you know, the MCU does not always execute instructions sequential, which is an issue we'll discuss later.

The program counter closely relates to two other pieces of hardware in the RAT MCU architecture. Because of this, we'll describe the other hardware in the context of the PC in hopes that it facilitates the understanding of the PC's basic functionality.

Figure 18-5 shows a high-level block diagram of the PC. The PC is truly a counter of some type, but it has some special features that the RAT MCU architecture requires for proper operation. You can describe the PC as a synchronous loadable up-counter with a few other special features. The output of the PC serves as the address input to the program memory. Table 18.1 shows a complete list and description of the program counter's inputs and outputs. The basic requirements of the PC are as follows:

- The PC needs to increment in order to handle sequential program execution as a simple increment is sufficient for all non-branching instructions.
- The PC must be able to load program memory addresses in order to support non-sequential program execution. Non-sequential program execution includes branch instructions, calling and returning from subroutines, and interrupt handling. Note this selection of loading inputs requires a MUX.
- The PC needs to be able to reset its count to 0x000.

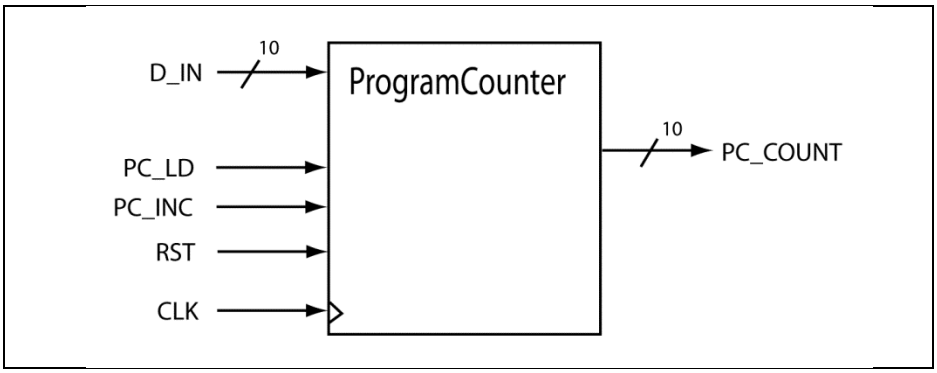


Figure 18-5: The block diagram for the RAT MCU's program counter.

Signal	Comment
PC_COUNT	A 10-bit signal that reflects the current value stored in the PC; the PC uses this value as an address to access instructions in the program memory.
PC_LD	Controls the synchronous parallel loading of data to the PC.
PC_INC	Increment control of the PC; when asserted, this signal allows the value in the PC to synchronously increment to support non-branching instructions. The PC_COUNT output does not change when this signal is not asserted.
D_IN	Data input for parallel loads of the PC.
RST	Synchronously resets the PC when asserted.
CLK	System clock

Table 18.1 Tabular explanation of the program counter inputs and outputs

Figure 18-6 shows the PC with a MUX attached to the PC's D_IN input. The PC must be able to parallel load one of three different values for non-sequential program execution. The MUX allows the PC to change according to the current instruction needing execution or in response to entering the RAT MCU's interrupt cycle.

- FROM_IMMED: Unconditional branch instructions, conditional branch instructions where program execution takes the branch, and call instructions each obtain the new value of the PC from the immediate value included as part of the individual instruction formats.

- FROM_STACK: Return-type instructions (RET, returns from subroutine while RETIE and RETID returns from interrupts) obtain the new PC value from the stack.

- 0x3FF: Interrupts automatically load the interrupt vector address to the PC when the RAT MCU enters an interrupt cycle. The current RAT MCU interrupt architecture uses the program memory address 0x3FF as the interrupt vector address.

The PC_MUX_SEL, PC_LD, and the PC_INC signals are control inputs to the PC. These signals are outputs from the control unit and are asserted according to the instruction currently being decoded by the control unit during the execute cycle. The PC_INC signal is only asserted during the fetch cycle.

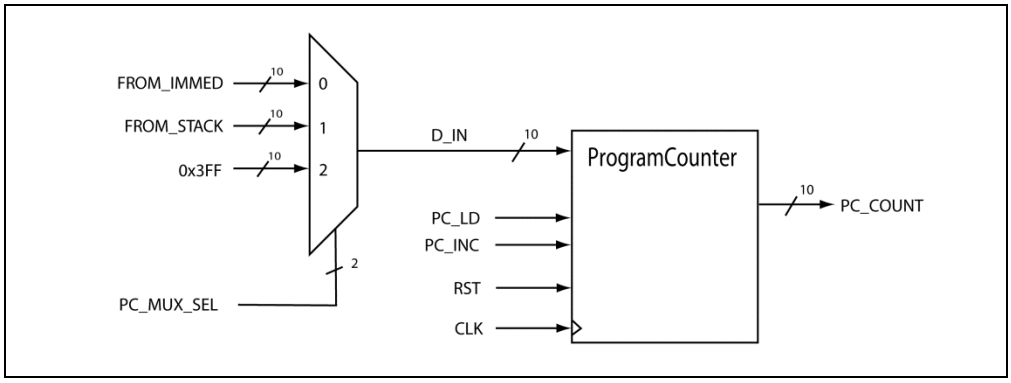


Figure 18-6: The program counter including PC input MUX.

We often refer to the RAT MCU's program memory as the "prog_rom" module, as it holds the program that runs on the RAT MCU. The prog_rom object is a 1024x18 synchronous ROM, which allows the associated program to have a maximum of 1024 instructions. The output of the PC forms the address input for the prog_rom, while the output of the prog_rom forms the opcodes and field codes for the individual instructions in the program.

Figure 18-7 shows the PC, the PC input MUX, and the prog_rom in a single diagram. There are two things to note about Figure 18-7.

- The outputs of the prog_rom are opcode and field code bits for the program's instructions. Figure 18-7 refers to the output of the prog_rom as the "IR", which stands for "instruction register". The notion of an instruction register is common in computer architectures, but the RAT MCU does not really have one. It's comfortable to model any MCU as having an instruction register, where the purpose of the instruction register is to "register" the outputs of the program memory for various purposes. In actuality, the outputs of the prog_rom are automatically "registered" as part of the prog_rom, which allows those so inclined to view the prog_rom as having an internal instruction register.

- Though Figure 18-7 does not show it, ten bits of the IR signal connect to the "FROM_IMMED" input of the PC MUX. These are the bits that associated with the immed operand in the in branch and CALL instructions. Recall that these types of instructions encode the branch address as part of the instruction bits in one of the instruction's field codes.

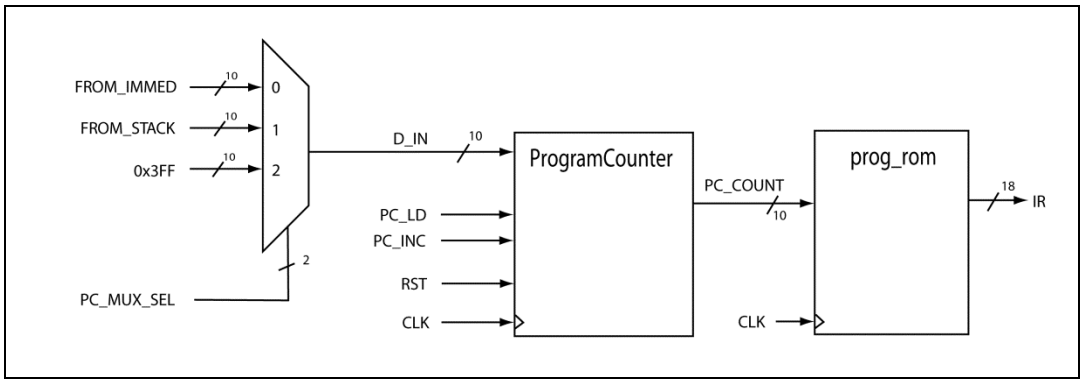


Figure 18-7: The program counter including PC input MUX and the program memory.

Figure 18-8 shows a high-level timing diagram showing the important signals associated with the program counter and program memory. Figure 18-8 shows two full instruction cycles. Here are a few other important items to note regarding Figure 18-8. The timing here looks a little strange, but if looks less and less strange the longer you stare at it[3].

- Though you may not see it from the Figure 18-8, the entirety of the fetch cycle entails one operation: incrementing the program counter. The control unit effectively increments the program counter by asserting the PC_LD signal upon entry to the fetch cycle. The program counter does not increment until the next clock edge, as the program counter is a synchronous device.

- The PC_ADDR line represents signal sent out from the program counter. Note that in the example below, the value in the PC_ADDR line represents an address in program memory; due to the characteristics of the PC_LD signal, the PC_ADDR increments at the start of every fetch cycle. This example shows "normal" operation where the PC is simply incrementing.

- The IR line represents the output from the program memory, which we sometimes refer to as an instruction register, or IR. Note the that RAT MCU officially does not have an IR; though we can think of the output of the program memory as a register since the program memory is a synchronous device, which means the output can only change on an active clock edge.

- The IR lines change at the beginning of the fetch cycle because of the rising clock edge of the fetch cycle and the previous change in the value of the address lines (PC_ADDR). Recall that the program memory is a synchronous device also.

- We include the CTRL_STATUS and STATUS_SIGNALS lines from the control unit timing for the sake of completeness. Once again, putting effort into understanding the entire timing diagram is a really good idea. It's really rather simple; it only seems tough because it's not overly intuitive.

[3] The key to realize here is that there is more than one way to properly "time" a computer. The RAT MCU has opted for one of these ways as we feel this way is the easiest to understand. We hope that as you gain more low-level computer design experience that you'll think of better ways to sequence computer operations.

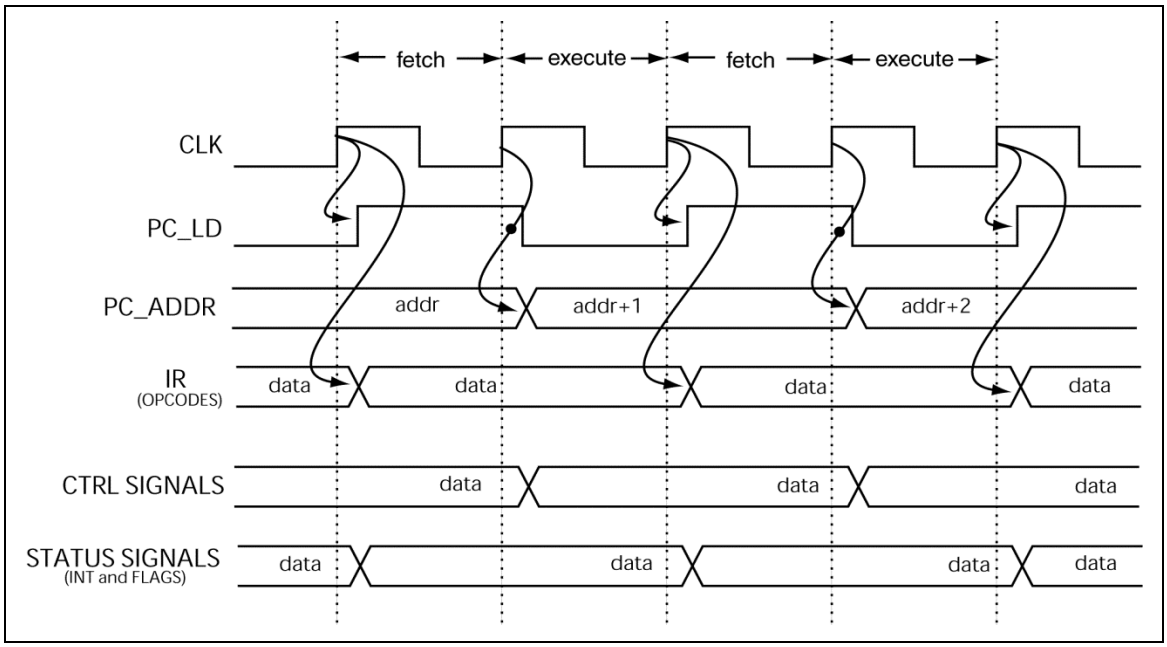


Figure 18-8: High-level timing diagram for the RAT MCU's program counter unit.

18.4 The ALU, the Register File and the Condition Flags

The register file, ALU, and condition flags form an important unit in the RAT MCU. These three modules are responsible for doing most of the bit-crunching in the RAT MCU, where the ALU does the bit-crunching, the register file provides the operands and store the results, and the condition flags provide information about the result of the ALU operation. The following sections provide details regarding these three modules.

18.4.1 The Register File

The register file provides storage for the operands associated with various operations in the RAT MCU. We refer to the register file as a 32x8 "dual-port RAM"; while the notion of a dual-port RAM could mean anything out in computer-land, it has a specific purpose in the RAT MCU. The dual-port RAM in the RAT MCU (in the worst case) must be able to read two operands and write one operand (for reg/reg-type instructions). The RAT MCU uses the two operands read from the register file as inputs to the ALU, while the write operand is a result from the ALU that the RAT MCU writes back to the register file.

Figure 18-9 shows a black box diagram of the RAT MCU's register file. Table 18.2 provides a description of the register file's input and output signals.

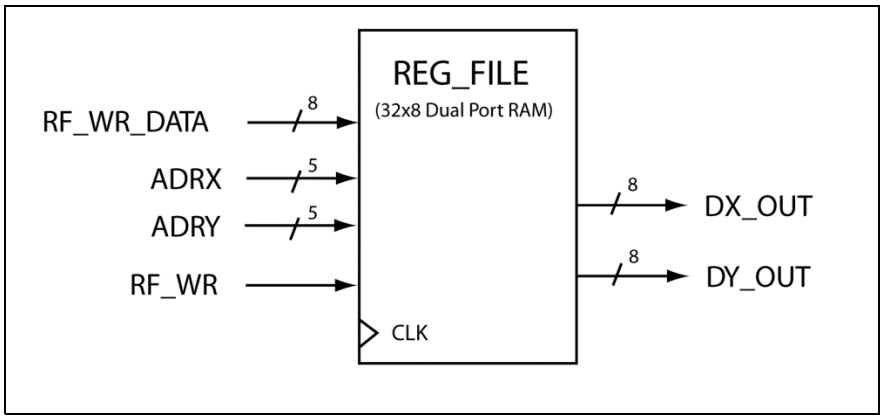


Figure 18-9: Black box diagram for the RAT MCU's register file.

Signal	Comment
RF_WR_DATA	The eight-bit input data to the register file. This signal is MUXed with other data signals, including the output of the ALU (the result).
ADRX ADRY	The address lines associated with the two register file inputs. These addresses are five bits in length, which provides access to the register file's 32 registers. The instruction register (output of the program memory) provides data for the ADRX and ADRY signals, which are five-bit field codes in the associated instructions.
DX_OUT	The 'X' or 'A' output of the register file. The DX_OUT data supplies data for scratch RAM writes (LD & ST-type instructions) and stack writes (PUSH & POP instructions).
DY_OUT	The 'Y' or 'B' output of the register file. This output is the source operand for ALU-type instruction, but can also provide address information for the indirect-type of LD and ST instructions.
CLK	The system clock; the register file is synchronous so this signal synchronizes register file operations.

Table 18.2 Tabular explanation of the RAT MCU's register file's inputs and outputs.

18.4.2 The Arithmetic Logic Unit

The RAT MCU's arithmetic logic unit (ALU) is a combinatorial module that implements the "data processing" type instructions in the RAT MCU's instruction set. The ALU can perform one of 15 different operations, where the result of these operations appears on the SUM output[4]. The C_OUT and Z_OUT provide information regarding the ALU operation; the control unit can latch this information into the C and Z flags based on specific requirements of any particular instruction. Table 18.3 provides a full description of the ALU's inputs and outputs.

[4] Yes, a better name for this output would be "RESULT".

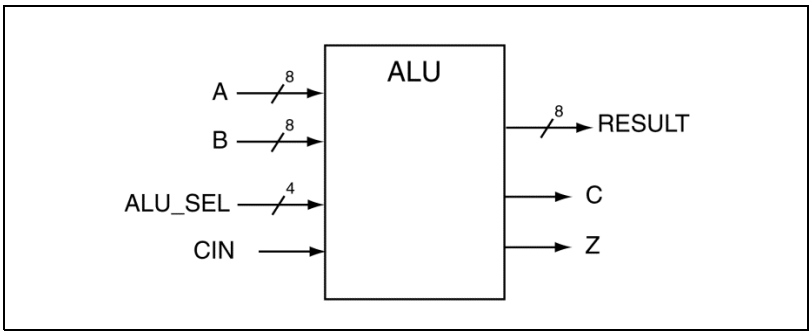


Figure 18-10: Black box diagram for the RAT MCU's ALU.

Signal	Comment
A,B	The two eight-bit operands to the ALU; these operands are outputs of the register file.
ALU_SEL	The selector signals the ALU uses to decide which functionality the ALU will perform. The ALU performs one of 15 different functions, which requires a minimum of four select signals.
C_IN	The "carry-in" input to the ALU; this input is the output of the storage element that the RAT MCU uses to store the C flag.
RESULT	The 8-bit "result" of the selected ALU operation.
C Z	Condition outputs from the ALU; these outputs are the inputs to the C and Z flag storage elements and provide information regarding the ALU operation. Different ALU operations affect these flags differently.

Table 18.3 Tabular explanation of the RAT MCU's ALU inputs and outputs.

18.4.3 The Conditions Flags

The RAT MCU uses what we refer to as condition flags to provide information regarding the results of various ALU operations. The condition flags on the RAT MCU include a carry flag (the C flag) and a zero flag (the Z flag). The C flag gives information regarding carry and borrow occurrences from addition and subtraction operations and serves as a single-bit source and receptacle for various shift and rotate operations. While the C flag serves several different purposes, the Z flag only serves one purpose: it specifies when the result of various ALU operations. Specifically, if the result is 0x00, then the Z flag is set to '1'; otherwise, it is set to '0'.

The RAT MCU represents the C and Z flags as single bits. Because the C and Z flag have persistent results, the RAT MCU represents these two flags using flip-flops. These flip-flops have load control inputs, which are direct outputs from the control unit. The input to these flip-flops can either be from the C and Z outputs from the ALU or from the shadow C and Z flags, respectively. The RAT MCU loads the C and Z flags from the ALU outputs except in the case of a context restoration operation as a result of issuing a return from interrupt instruction (RETIE or RETID). This being the case, there are two other condition flags we refer to as the shadow flags, which the RAT MCU uses to save and restore operating context associated with interrupt processing. We'll discuss interrupt processing in a later section of this chapter.

18.4.4 ALU, Register File, and Condition Flag Circuitry

Figure 18-11 provides a diagram of the register file, ALU, and condition flag circuitry. This diagram does not include all of the circuitry, but it does show the more important signals involved. Here are some of the more important points regarding Figure 18-7:

- The RF_WR_DATA signal provides the data to write to the register file. This can come from various sources including scratch RAM, the I/O unit, or the result of the ALU operation. The control unit controls a MUX (not shown) which handles these various inputs.

- The output of the instruction register (the program memory output) provides the data for the ADRX and ADRY signals. These five-bit values are associated with the register field-codes in the RAT MCU's instructions.

- The DX_OUT output of the register file connects directly to the ALU. DX_OUT also connects to the scratch RAM data input and the RAT MCUs data output signal, but we'll save those for another discussion.

- The DY_OUT is MUX together with an immed value from the instruction register (the output of program memory). This allows the RAT MCU's control unit to use the REG_IMMED_SEL input to choose the second operand to the ALU. Recall that the RAT MCU assembler encodes the immed value one of the field codes in that type of instruction.

- The ALU has four inputs: two operands, a carry-in, and the ALU operation select inputs. The ALU uses the first three inputs for calculations (data inputs) and the select inputs for control. Recall that the ALU is a combinatorial device.

- The condition flag module consists of two flip-flops, one for the C flag and one for the Z flag. The system clock synchronizes state changes with these modules. The diagram does not show the Z output directly.

- The output of the C flag is the input the to the C_IN input of the ALU.

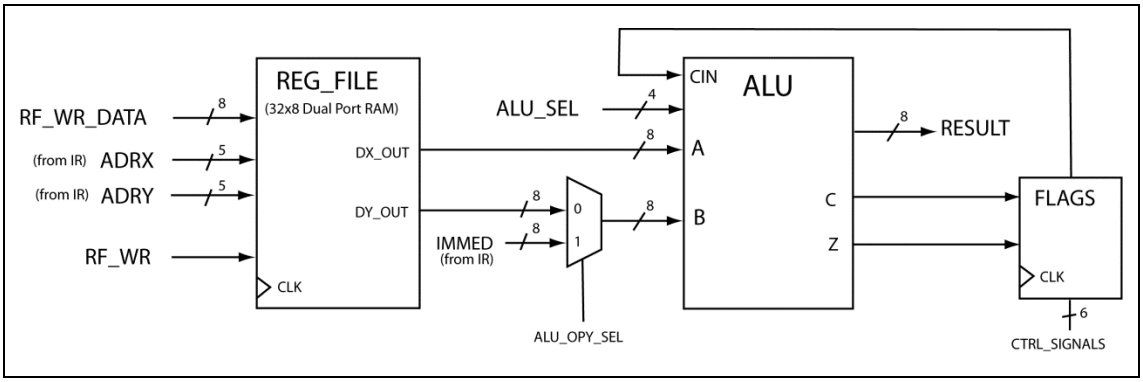


Figure 18-11: Black box diagram for the RAT MCU's bit-crunching circuitry.

Figure 18-12 shows an expanded diagram of the FLAGS module from Figure 18-11. The FLAGS module in Figure 18-12 is not complete, as it does not include the shadow flags; we'll describe the shadow flags in a later section of this chapter. Here is some important information regarding Figure 18-12.

- As you can see from Figure 18-12, we model the C and Z flags as D flip-flops. The C flag is slightly more complicated than the Z flag as it contains both set and clear inputs. The C flag requires these inputs but the Z flag does not because there two RAT instructions that directly manipulate the C flag (SEC and CLC).

- The diagram does not show the CLK line in order to make the diagram as clear as possible. The two flip-flops connect to the system clock.

- Both of the flag outputs feed back to the control unit inputs. These inputs thus serve as status inputs to the control unit. The C flag feeds back to the input of the ALU and allows it to serve as input to several instructions.

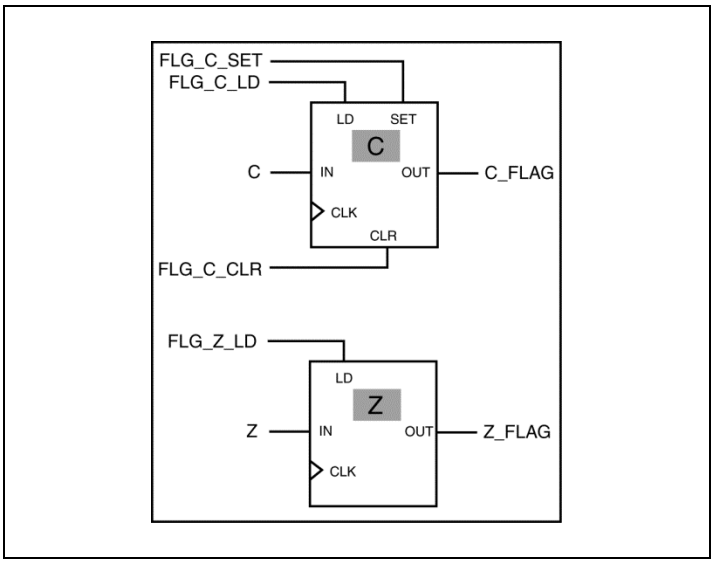


Figure 18-12: A diagram of the FLAGS box showing the C and Z flags (this diagram does not show the shadow C and Z flags).

18.5 The Scratch RAM

The RAT MCU architecture includes a "scratch RAM" module. This 256x10 memory is one of three[5] memory units in the RAT MCU architecture and serves several specific purposes. Computers generally have different memory architectures depending on the need of that particular architecture. The RAT MCU architecture is a relatively simple[6] general-purpose architecture. Thus, the RAT MCU has a modest program memory size, a modest register file size, and, there is a modest amount of extra memory that the RAT MCU can access by directly by some instructions and access indirectly by other instructions.

The term "scratch" relates to "scratch paper" where you may temporarily jot down a phone number. In computer architectures, the scratch RAM stores intermediate results and/or values you may need to access multiple times. These two issues are distinctively different.

1. Intermediate results are values from an ongoing calculation that you may not have room in your register file to store. The only downside of using the scratch RAM to store values is that they require more time (instructions) to access as opposed to values stored in the register file.

[5] The other two memory units include the register file and the program memory.
[6] Compared to other available MCUs.

2. Values that the progrom accesses often are often stored in scratch RAM. We generally refer to this type of storage as a look-up table (LUT).

The notion of a scratch RAM refers to an intermediate storage device that is both readable and writeable. We model the RAT MCU's scratch RAM as a 256x10 RAM. Figure 18-13 shows a black box diagram of the RAT MCU's scratch RAM. The RAT MCU's scratch RAM serves to main purposes: 1) general data storage, and 2) the stack. This being the case, the scratch RAM memory must be able to do the following:

- The scratch RAM must write data to and read data from any address location in the scratch RAM. This functionality supports the LD and ST instructions. Recall that data written to the scratch RAM via the ST instruction necessarily comes from a register in the register file and data read from the scratch RAM via LD instruction writes to a register in the register file. These writes and reads are 8-bits only.

- The stack functionality of the scratch RAM must be able to write addresses or register data to and read address and register data from the scratch RAM. The address data is 10-bits wide and is associated with CALL and RET-type instructions. The register data is 8-bits wide and is associated with PUSH and POP instructions.

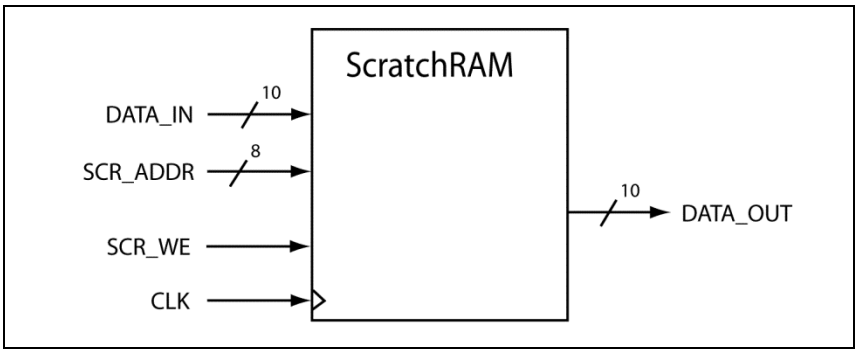


Figure 18-13: Black box diagram for the RAT MCU's scratch RAM.

Signal	Comment
SCR_ADDR	The address inputs to the scratch RAM.
SCR_WE	Controls the synchronous parallel loading of the SCR_DATA input data to the PC.
DATA_IN	The 10-bit input data to the RAM. This input comes from either the PC or DX output of the register file.
DATA_OUT	The 10-bit data output from the RAM. This data is either sent to the PC or the register file.
CLK	System clock input; synchronizes scratch RAM operations.

Table 18.4 Tabular explanation of the scratch RAM's inputs and outputs

The scratch RAM in the RAT MCU architecture serves two main purposes: general purpose storage and as a stack. The RAT MCU architecture uses the scratch RAM as storage for the stack. Using a scratch RAM device for both intermediate storage and a stack is common in computerland because it means you need less "memory" devices in your architecture. The downside of having the stack being part of the scratch RAM is that your stack can overwrite useful data if you don't control the depth of subroutine nesting or the number of the number of pop-less pushes or push-less pops. Another downside of using the same device for intermediate storage and a stack is that you can overwrite return-type data under program control using store (ST) instructions.

18.5.1 The Stack Pointer

The stack point is a register that the RAT MCU uses to store the memory location of the top-of-the stack. The RAT MCU implements the stack pointer as a synchronously loadable up/down counter. Recall that counters are a special type of register. The RAT MCU uses the stack pointer output as one of several selectable address inputs to the scratch RAM, which it uses in instances where the executing instruction uses the stack. Since the stack pointer must address one 256 location in scratch RAM, the stack pointer must be 8-bits wide. Figure 18-14 shows a black box diagram of the stack pointer while Table 18.5 provides a description of the stack pointer's inputs and outputs.

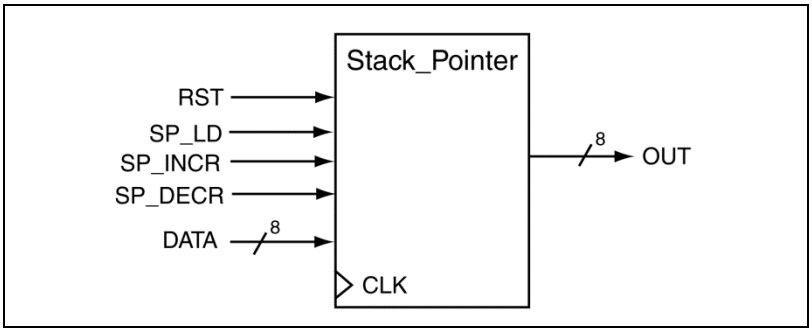


Figure 18-14: Black box diagram for Stack Pointer.

Signal	Comment
RST	A synchronous reset input to the stack pointer; when asserted 0x00 is loaded into the stack pointer.
SP_LD	Parallel load control input to the stack pointer. When this input is asserted, the signals on the DATA inputs are synchronously latched into the stack pointer.
SP_INCR	Increment control input to the stack pointer. When this input is asserted, the value currently stored in the stack pointer increments (increases by 1) on the next active clock edge.
SP_DECR	Decrement control input to the stack pointer. When this input is asserted, the value currently stored in the stack pointer decrements (decreases by 1) on the next active clock edge.
DATA	Data input to the stack pointer; this input connects to the register file's DX_OUT, which provides support for loading a value into the stack via a register.
OUT	The output data signals for the stack pointer, which serve as one of the address inputs to the scratch RAM.
CLK	System clock;sSynchronizes stack pointer operations.

Table 18.5 Tabular explanation of the stack pointer's inputs and outputs

Figure 18-15 shows a diagram of all the important modules associated with stack operations. Here are few important things to note about this figure.

- The scratch RAM can use one of four addresses to access the scratch RAM; the control unit selects which of these addresses to use based on the instruction the RAT MCU is executing. The four options are:

    1. From the register file: LD and ST instruction using indirect arguments use this option because the register in the register file stores the address value.

    2. From IR: LD and ST instructions can also use immed values; the RAT assembler encodes these values in the immed field code of the associated instruction.

    3. From the stack pointer: the control unit chooses this address for the stack operation associated with pops, returning from subroutines, and returning from interrupt processing[7].

    4. From the stack pointer [-1]: the control unit choose this address for stack operations associated with pushes, subroutine calls, and the execution of the interrupt cycle.

- The box containing the "-1" indicates that the value that is MUXed is one less than the current value in the stack pointer. While this is handy to model as a "-1", this hardware is actually an 8-bit adder. This value must be in place to assure that the push-type stack operations (push, subroutine calls, and initial interrupt processing) write to the appropriate location in the stack. The preferred RAT MCU stack implementation points at the last object placed on the stack (which we roughly define as the top of the

[7] Some of this information may be beyond what you know now, but we'll get to these topics later in this chapter.

stack[8]). This means that push-type operations must write to one address before the current start pointer address. It is not possible to decrement the stack and then write the new value to the stack in one clock cycle, so the alternative the RAT MCU uses is to use an adder to decrement the current stack pointer value and always have that value available. Somewhat kludgy, but it works.

- The RST, SP_LD, SP_INCR, SP_DECR, SCR_WE, and SCR_ADDR_SEL signals are all control signals and are thus direct outputs of the RAT MCU's control unit. The control unit uses the correct combination of these signals in order to implement the operations associated with various instructions and other RAT MCU functionality.

- The diagram does not show the CLK line in order to make the diagram as clear as possible. The two devices share the system clock signal.

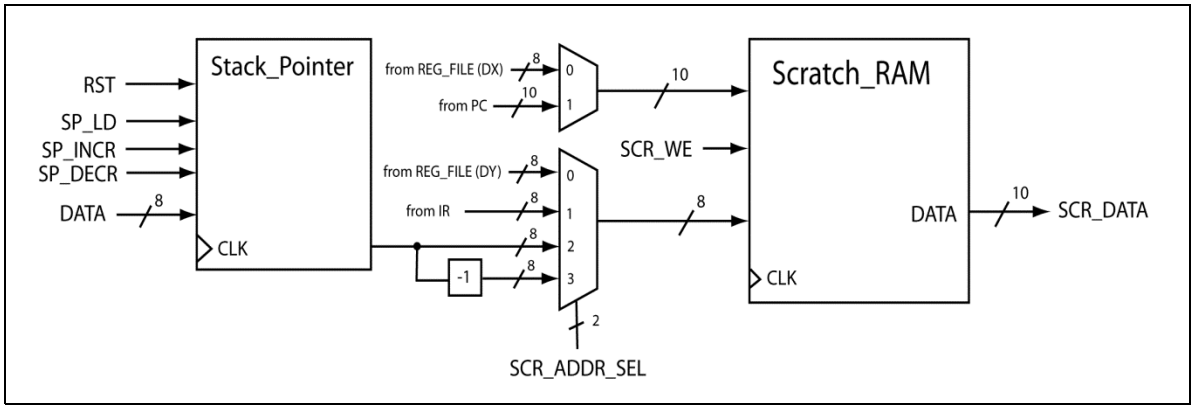


Figure 18-15: Black box diagrams for the "stack" including both the stack pointer and the scratch RAM.

18.5.2 PUSH and POP Instructions

PUSH and POP instructions provide the programmer with a way to store bytes of data onto the stack. There are several approaches to implementing a stack in a microcontroller. We have arbitrarily chosen the stack to be part of the scratch RAM in the RAT MCU. Additionally, there are several approaches to implementing push and pop operations on the stack. We opted to use the most common method with the RAT MCU. The overall goal is to understand and implement a stack object in hardware; if you can understand one approach, the other approaches will seem trivial. We'll only describe one approach in this section.

Figure 18-16 provides both a written and RTL-based description of the RAT MCU hardware responsibilities associated with the PUSH and POP instructions. Keep in mind that that the two operations listed for both the PUSH and POP operations occur on the same clock cycle; they are part of the execution cycle of the control unit. Here are a few more important items to note regarding Figure 18-16.

- The use of parenthesis indicates indirection. In computer terms, this means that data is not taken from the value in the parenthesis; it means the value in parenthesis is an address and data is taken from the location that this address specifies.

- The RAT MCU considers the top-of-the-stack to be pointing at the previous valid item that the hardware placed onto the stack. This means that for push operations, the hardware must write the new data to the next valid location on the stack, which is one less than the current value of the stack pointer.

[8] Recall there are different approaches on this. The approach we take is where the SP points to the last item written to the stack. We also could have had the SP point at the next empty location after the last item written to the stack.

This is arbitrary; we could have chosen to do this differently, but most commonly the stack pointer value decreases with pushes and increases with pops.

- Due to the previous bullet, the SP is pointing at the correct data that needs removing from the stack for pop operations. Note the different between the push and pop operation.

- The notion of the SP decrementing for push operations and incrementing for pop operations is arbitrary. The approach we use for the RAT MCU is the most common approach with the stack pointer becoming a smaller value with continued pushes. This is another reason why most microcontrollers give you the option of writing a value to the stack pointer, which the RAT MCU also does with the WSP instruction.

PUSH Operation	POP Operation
• data is written to one location less than what the current stack pointer is pointing at • stack pointer is decremented	• data is read from address pointed to by stack pointer • stack pointer is incremented
RTL Notation	**RTL Notation**
(SP-1) ← data_val SP ← SP - 1	data_val ← (SP) SP ← SP + 1

Figure 18-16: Details of push and pop stack operations in words and RTL notation.

18.5.3 LD and ST Instructions

The LD and ST instructions allow programmer direct access to any data location in the scratch RAM, with the LD instruction being a memory read and the ST instruction being a memory write. Unlike the PUSH and POP instructions, the LD and ST instructions do not involve the stack, which directly implies the main difference between the LD/ST instructions and PUSH/POP is that the LD/ST instructions can access any value in the scratch RAM while the PUSH/POP instructions can only access data associated with the stack pointer.

Figure 18-17 shows the general form of the LD and ST instructions. There are a few items of importance in Figure 18-17Figure 18-17, which are:

- All operations associated with the LD and ST instructions occur through a register. For memory write operations, the register holds the data the programmer wants to write to scratch RAM; for memory read operations, the hardware writes the data read from scratch RAM into the register operand. This means there must be a data path from the output of the register file to the data input of the scratch RAM for ST instructions and a path from the output of the scratch RAM to the input of the register file for LD instructions.

- The addr operand represents an 8-bit value that specifies the address in scratch RAM to read from or write to. The “addr” operand can come from one of two sources. Programmers can specify the value directly by providing an immediate value or indirectly by specifying register whose contents contains the address value.

```
LD      reg,addr
ST      reg,addr
```

Figure 18-17: The general form of the LD and ST instructions.

Table 18.6 shows the two forms of the LD and ST instructions. For the reg/immed forms of the instructions, the hardware uses the immed field code from the instruction bits as the address to the scratch RAM. For the reg/reg form of the LD and ST instructions, the register field code in the instruction format indicates the address of the register that holds the address to access scratch RAM. In this case, the hardware chooses the output of the register file as the address input to the scratch RAM. In both cases, the control unit is responsible for supplying the appropriate control signals to route the data to its required locations.

Instruction Type	Usage Example	Comment
LD reg/immed	`LD r0,0x33`	Writes the value in register r0 into scratch RAM location 0x33
LD reg/reg	`LD r0,(r2)`	Writes the value in register r0 into scratch RAM location specified by the value in register r2 (r2 is treated as an address)
ST reg/immed	`ST r0,0x33`	Reads the value in scratch RAM location 0x33 into register r0.
ST reg/reg	`ST r0,(r2)`	Reads the value in the scratch RAM location specified by the value of register r2 into register r0 (r2 is treated as an address).

Table 18.6: Examples LD and ST instruction-types.

One important item worth noting here is that the width of the scratch RAM data is 10-bits, while the width of register data is 8-bits. The RAT MCU uses the scratch RAM for both general purpose storage as well as the stack. Because part of the scratch RAM is also the stack, the scratch RAM data must be 10-bits wide as the hardware uses the stack to store program memory address. Recall that program memory contains 1024 locations, which subsequently requires ten bits to access.

18.6 Input/Output Architecture

The input/output is one of the three main subsystems of the basic compute model. It makes intuitive sense that for a computer to do anything useful, you need to give it some input and the computer needs to return a result via the output. The I/O "subsection" on the RAT MCU is relatively simple and thus we cannot model it as being a typical "block module" as we did with other RAT architecture subsections.

The IN and OUT instructions handle all of the details regarding the RAT's input/output mechanism. As you can see from examining Table 18.7, all input and output with the RAT MCU involves the register file. Specifically, the IN instruction writes input values to the designated register while the OUT instruction makes data from a designated register available to the outside world.

Instruction	RTL	Example
IN	Rd ← in_port(imm_val)	`IN Rd,imm_val`
OUT	out_port(imm_val) ← Rd	`OUT Rd,imm_val`

Table 18.7: Examples of the RAT MCU I/O instructions: IN and OUT.

Figure 18-18 shows a high-level model of the RAT MCU. A quick examination of Figure 18-18 shows that most the RAT signals are part of the RAT's input/output mechanism, where only the only the CLK and RESET signals are not.

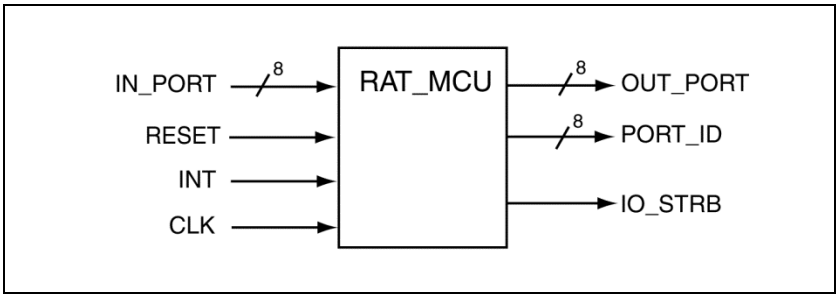


Figure 18-18: The RAT MCU black box diagram.

The hardware operations associated with the IN and OUT instructions are relatively simple in terms of the underlying RAT MCU hardware. Here are the pertinent points regarding the two instructions and their relation to the underlying hardware.

- The input and output instructions are both register-based instructions. The IN instruction writes data from the outside world to a register, while the OUT instruction writes the value in a particular register to the outside world.

- Bothe the IN and OUT instructions are reg/immed-type instructions. The immed-type operand is one of the field codes that form the instruction bits for the instruction. The underlying hardware makes these bits available to the outside world when the one of these instructions execute. These bits become available as the PORT_ID output of the RAT MCU.

- Hardware external to the RAT MCU will necessarily use the PORT_ID bits in such a way as to support the notion of 256 different PORT_IDs. This external hardware lives in the RAT MCU "wrapper"; the notion of a wrapper entails all the hardware required to interface the RAT MCU with external peripherals, and in particular, the inputs and outputs associated with those peripherals.

- The external input data connects to the IN_PORT input of the RAT MCU. Because the IN instruction is register-based, the input data connects to the register file via the register file MUX. The control unit is thus responsible for issuing the proper MUX control signals when the RAT MCU executes an IN instruction.

- The external output data connects to the DX_OUT output of the register file.

- The outside world needs to know when the RAT MCU executes an OUT instruction. This is because the output values of the RAT MCU must be registered since the output values are only guaranteed to exist for the duration of the execute cycle. The RAT MCU uses the IO_SRTB signal to generate a

pulse as part of the OUT instruction. The IO_STRB output of the RAT MCU is a direct output from the control unit.

Figure 18-19 shows a timing diagram associated with sample IN and OUT instructions. The diagram shows the pertinent signals involved in and their characteristics when the IN and OUT instructions execute including both the fetch and execute cycles. Here are the important features of this diagram.

- The PC address increments as the MCU enters the execute cycle; the increment control on the PC asserts as part of the fetch cycle.

- The bits associated with the instruction become available the clock cycle after the address of the instruction becomes valid. The port_id associated with the IN and OUT instruction is one of the fields in the opcodes bits and is thus available at this time.

- The IN instruction reads from the IN_PORT. We generally consider this data on the IN_PORT lines to be asynchronous, which is why we opted to change the input data bits at no particular time. The IN_PORT lines connect to the register file; control signals output from the control unit ensure the IN_PORT data is latched into the register specified by the IN instruction.

- The OUT instruction outputs data from the register file to the outside world. The actual data from the register file becomes available as part of the execute cycle of the OUT instruction. The register data is only valid for the duration of the OUT instruction's execute cycle; after that the RAT MCU treats the data as don't cares.

- The RAT MCU's hardware generates the IO_SRTB pulse during the execute cycle of the OUT instruction. The IO_STRB signal asserts because of the control unit decoding the instruction bits associated with the OUT instruction.

- The diagram uses "??" to indicate the values are not known or do not matter.

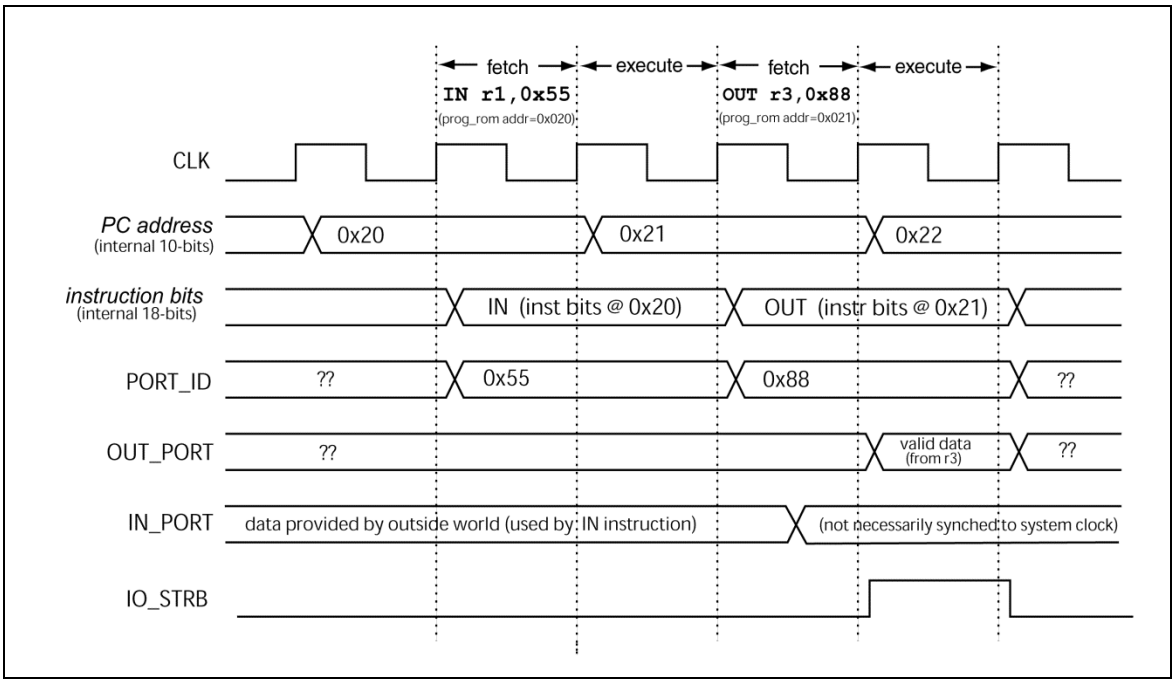


Figure 18-19: A timing diagram showing the pertinent characteristics of the IN and OUT instructions.

18.7 Program Flow Control Operations

Program flow control refers to the flow of instruction execution for a given programs. Programs generally execute instructions in a sequential manner in particular blocks of instruction code. Programs that only executed instructions sequentially would be really boring, so programs generally jump from one block of code to another. We classify this "jumping around" as program flow control.

To be precise, program flow control refers to any instruction or operation that causes the program counter to change in any manner other than sequentially[9]. There are three types of program flow instruction/operations on the RAT MCU: branch instructions (including unconditional and conditional branching), subroutines associated instructions, and interrupts. We discuss the first two items in this section; we discuss interrupts in the following section.

18.7.1 Branch Instructions

There are two types of branch instructions: unconditional branches and conditional branches. Both types of instructions are similar in that they include a field code in the associated instruction formats that the RAT MCU hardware uses as the branch address (the address of the instruction to execute if the code takes the branch). The branch address is a 10-bit value that the RAT MCU hardware latches into the program counter under the appropriate conditions. The unconditional branch instruction (BRN) always loads the value in its immed field code into the program counter when the instruction executes. As the names states, the conditional branch instructions (BREQ, BRNE, BRCC, and BRCS) only load the immed field-code into the program counter when the conditions are correct.

The RAT MCU has two condition flags: the C flag and the Z flag. The conditional branch instructions base the notion of their branching, or the loading of their immed field-code into the program counter, based on the state of

[9] By sequentially, we mean that it increments by one.

either the C or Z flag. There are two possibilities in this scenario. If the conditions are met, the RAT MCU hardware loads the branch address into the program counter; otherwise, the program counter simply increments causing the RAT MCU to execute the instruction following the conditional branch instruction. A common programming term is that we say the program flow "drops down" or 'drops through" past the conditional branch instruction if the instruction does not take the branch. Note that it is the responsibility of the control unit to ensure the instruction does the correct thing based on the given condition. This fact highlights the notion that the C and Z flag inputs to the FSM actually do something useful in their role as "status" inputs.

18.7.2 Program Flow Control: Subroutines

Subroutines represent another form of program flow control-type instructions in the RAT MCU instruction set. The notion of a subroutine means that that program execution "jumps" to another location in program memory in order to execute some set of instructions; the RAT MCU uses a CALL instruction for this jump. When the subroutine is complete, the RAT MCU returns to the instruction following the CALL instruction by issuing a "RET" instruction. The CALL and RET return instructions both utilize the stack in their operations. Recall that PUSH & POP instructions also utilize the stack. Interrupts also indirectly tweak the stack; we'll discuss interrupts in a later section.

As a refresher to what the CALL & RET instructions actually do, check out the written description in Table 18.8. A more technical description is sure to follow.

Event	Description	The Details
CALL instruction	Control of the program transfers to the instructions associated with the subroutine.	The PC, which has already been incremented, is pointing at the instruction in memory that will execute after the CALL instruction. The hardware pushes the address of this instruction onto the stack; the hardware then decrements the stack pointer.
RET instruction	The code associated with the subroutine has completed execution. Program control returns to the instruction following the original CALL instruction.	The stack is popped into the PC. The address pointed to by the stack pointer contains the address of where the program should begin executing after the subroutine has completed execution. This return address was pushed onto the stack when the CALL instruction was executed. The hardware then increments the stack pointer.

Table 18.8: A low-level description of the subroutine calling mechanism.

When the RAT MCU executes the CALL instruction, the hardware pushes the value representing the address of the next instruction after the CALL instruction onto the stack. Recall that in the RAT architecture, the PC increments at the end of the fetch cycle and is already pointing at the next instruction in memory that normally would execute next. The RAT MCU has a ten-bit wide stack because the stack must be able to store a program memory address. The stack can have up to 256 storage locations due to the fact the RAT MCU uses the scratch RAM for its physical implementation.

The address data that the RAT MCU stores on the stack as a result of a CALL instruction is the address of the instruction that follows the CALL instruction. This is thus the instruction that should be executed after the RET instruction in the subroutine executes. In other words, the CALL instruction places the address of where the subroutine instruction should return to onto the stack (a push operation). Note that the hardware places the pushed data at the address one less than the SP. The RET instruction removes the address value from the stack

(a pop operation) and places it into the program counter. This forces the next instruction to be executed after the RET to be the instruction following the CALL instruction that originally invoked the subroutine.

In order to allow the RAT MCU hardware to execute all instructions in two clock cycles, either the CALL operation or the RET operation must do "something special". The official description of the top-of-the-stack in the RAT MCU is that the stack is pointing at the most recently item popped onto the stack. This works fine for RET instructions, but in order for CALL instructions to operate in two clock cycles, the hardware must push the return address onto the stack at one location less than the value of the current stack pointer. This is an arbitrary decision; we could have opted to do this "special operation" with the RET instruction instead. There are many ways to officially implement a stack in a MCU; we chose the one we're describing here for no particular reason (so don't try to figure out the logic behind it).

Table 18.9 shows the underlying details of the CALL and RET instructions; RTL equations can easily describe the operations associated with the CALL and RET that the underlying hardware is responsible for completing as part of the instructions. From examining Table 18.9 and noting the following notables, you'll be able to absorb the importance of always having your CALLs and RETs happen in pairs, because if they do not occur in pairs, you will degrade the integrity of the information on the stack. Additionally, here are some other important things to note regarding the RTL equations in Table 18.9.

- The SP is the stack pointer and points to the last item pushed onto the stack in the RAT MCU.
- For a CALL instruction, the parenthesis around "SP" indicates indirection. In other words, the hardware does not write the PC to the SP, the hardware writes the PC to the location in the stack that the SP is pointing at; to be accurate with the CALL instruction, the hardware writes the PC to one less than what the SP is pointing at. Use of parenthesis to indicate this type of indirection is common in computerland.
- The PC increments at the end of the fetch cycle, which means that it is pointing to the next location in program memory during the execution cycle. This means that when the CALL instruction pushes the PC onto the stack, it actually pushes the address in program memory of the instruction that should execute following the RET instruction execution associated with the subroutine.
- The *label_addr* is the address of the first instruction in the subroutine. For the CALL instruction, the hardware places this address into the PC, which ensures that it is the next instruction the hardware executes after the CALL instruction. This address will be taken off the stack when a RET is executed in the subroutine. The label_addr officially appears in the program as text; the assembler changes the text to a valid address based on given program parameters.
- The first two operations listed in the CALL RTL statement represent the stack push operation. The two RTL statements in the RET describe a pop operation.
- All of the listed RTL statements for the CALL and RET instructions occur in one clock cycle.

CALL	RET
(SP - 1) ← PC SP ← SP - 1 PC ← label_addr	PC ← (SP) SP ← SP + 1

Table 18.9: The RTL statement describing CALL and RET instructions.

Figure 18-20 provides yet another explanation of the CALL and RET instructions. This explanation uses part of the main RAT MCU architecture diagram to show the call and return mechanism in terms of the existing RAT MCU hardware. The notes in Figure 18-20 are basically self-explanatory, so we won't blather on here.

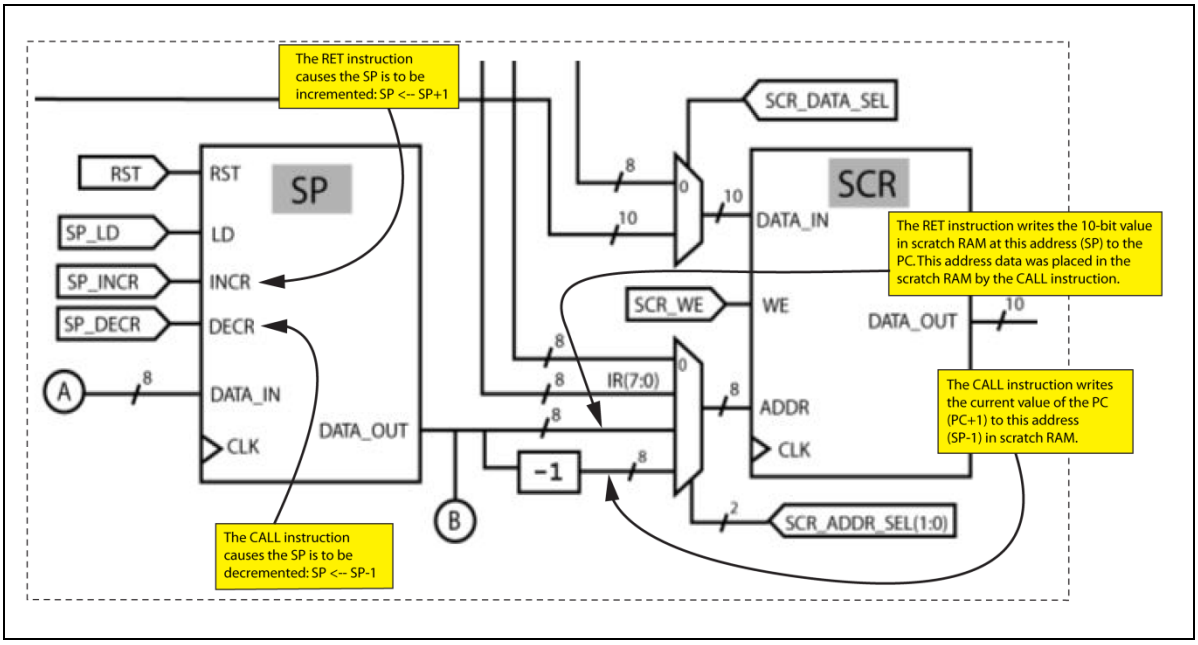


Figure 18-20: A visual and written explanation of subroutine calls and returns.

Figure 18-21 and shows a timing diagram with select signals for a CALL instruction. Figure 18-22 shows a similar timing diagram for the corresponding RET instruction. Each of these figures contain all the pertinent notes for the diagrams, so we'll avoid repeating them here.

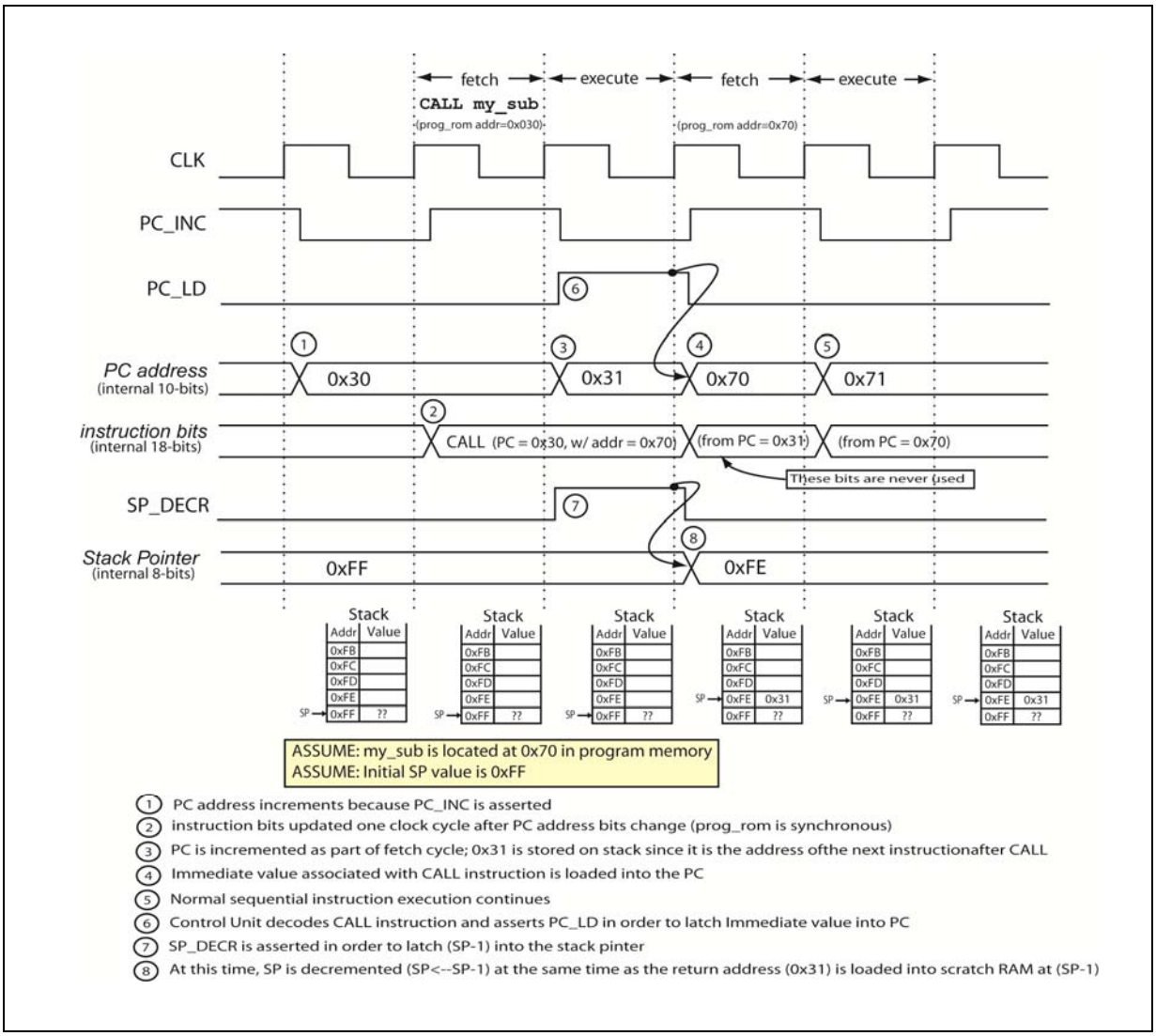


Figure 18-21: A partial timing diagram a for CALL instruction.

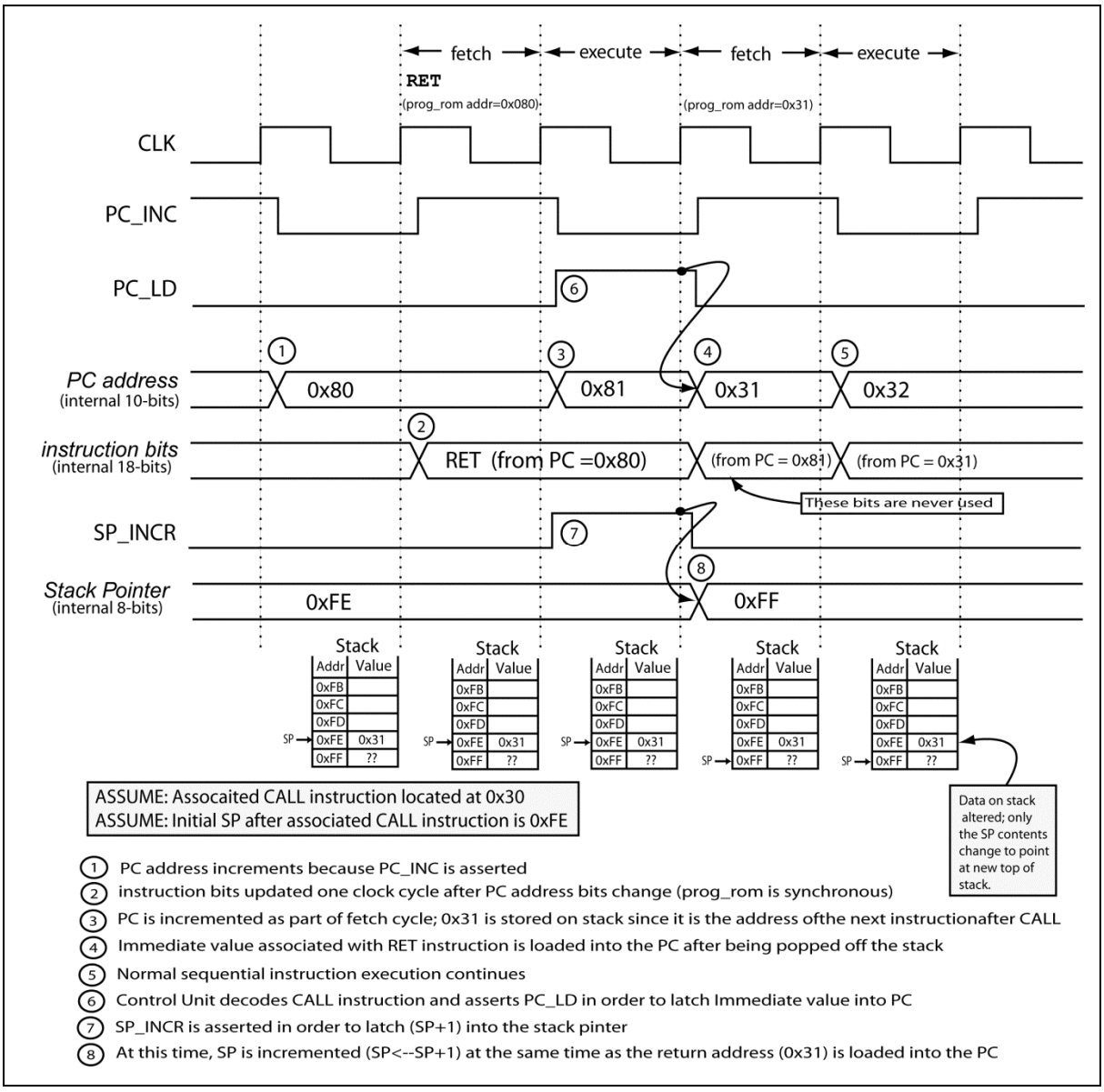


Figure 18-22: A partial timing diagram showing a RET instruction.

18.7.3 Program Flow Control: Interrupts

The notion of interrupts form the heart of most embedded systems. The interrupt mechanism essentially provides hardware with the ability to "call" subroutines. Moreover, interrupts give the hardware the ability to react to system inputs, and this reaction time is much shorter than if you did not use interrupts.

Because interrupts are a form of "subroutine execution", they necessarily alter the normal flow of instruction execution, so they are associated with program flow control. We even give it the name "interrupt" because the normal program flow is "interrupted" to do some other special processing. Interrupts provide a method allow other computer peripheral devices to "call" a special subroutine. Interrupts are an important part of most every microcontroller and microprocessor. This being the case, we'll cover the RAT MCU's interrupt architecture in another section of this chapter. We place a mention of interrupts in the program flow section to remind the reader that interrupts truly affect program flow control in typical microcontrollers.

18.8 RAT Interrupt Architecture

The concept of interrupts is relatively simple due to their similarity to subroutines. You can generally connect external peripheral devices to an MCU in such a way as they can do what we refer to as "generate an interrupt". While this may sound complicated, it simply means that the device has the ability to make a signal high; this signal is understood to be connected to the interrupt input on the MCU. When an external peripheral device "generates an interrupt", the microcontroller stops what it is doing and starts processing a special subroutine known as the *interrupt service routine*, or ISR. When processing of the ISR is complete, the microcontroller resumes to its regularly scheduled programming. Simply stated, the ISR is nothing more than a subroutine initiated by hardware. The section covers the details of how the RAT hardware processes the interrupts.

Because interrupts are similar to subroutines, the hardware portions of the RAT MCU's interrupt architecture is minimal. Because the hardware already support subroutines, most of the required hardware for interrupt processing is already in place. The hardware that is missing handles details such as interrupt masking and context saving/restoration, which we describe in a later subsection of this chapter.

18.8.1 The Interrupt Cycle

The RAT MCU has a special input, which we refer to as the interrupt input. The RAT MCU architecture refers to this external input as the "INT" input. Some external peripheral device uses this input to generate interrupts on the RAT MCU. In terms of the underlying control hardware in the RAT MCU, we refer to the process of acting on an interrupt as the RAT MCU entering an "interrupt cycle". Figure 18-23 shows the complete FSM for the RAT MCU including the so-called interrupt cycle. The following items list the high-level details of the interrupt cycle as it relates to the RAT MCU's FSM.

- The FSM shows that at the end of the execute cycle, the FSM can either go to the fetch state or enter an interrupt cycle. Two conditions must be present in order for the FSM to go into an interrupt cycle. First, the interrupt must be unmasked (enabled), and second, the INT input on the RAT MCU must be asserted. When these two conditions are present, the FSM transitions to the "interrupt" state in the FSM.

- The interrupt state in the FSM causes the hardware to execute the various operations associated with the interrupt cycle. We'll save the details for a later section, but what happens in the interrupt state is that the control unit sends out the control signals that implement the various operations associated with the interrupt processing. Once again, this processing is similar to the processing of a subroutine call.

- The interrupt cycle we speak of is associated with "going into an interrupt"; note that there are no special FSM states associated with exiting an interrupt cycle. As you'll see soon, exiting an interrupt is a matter of issuing a RETIE or RETID instruction; the RAT MCU hardware processes these instructions in a normal manner. These two instructions are similar to a normal return instruction, but do a few other things that we'll describe later.

- As the state diagram shows, it appears possible that the FSM can go immediately back into an interrupt cycle after it receives an interrupt. For reasons you'll see later, one function of the RAT interrupt architecture prevents this from occurring.

- The RAT MCU happens to have an interrupt cycle comprised of a single state. In general, the amount of "stuff" that needs to be done to support the interrupt architecture determines the length of the interrupt cycle. The RAT MCU happens to be able to do everything it needs to do to implement the interrupt architecture in a single state; this would not necessarily be true of other MCUs.

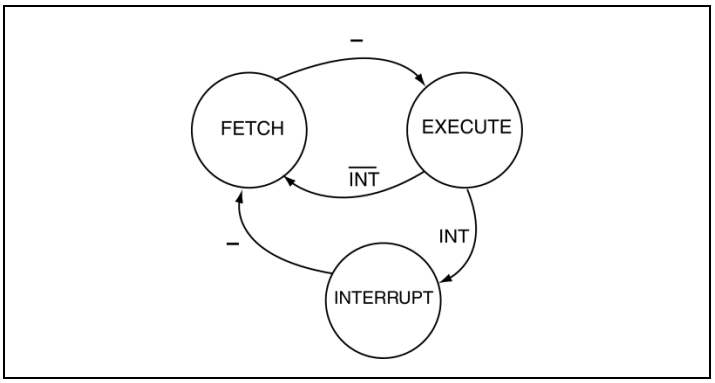


Figure 18-23: The state diagram for the RAT MCU showing the interrupt cycle.

18.8.2 Important Interrupt Timing Issues

Typical system clock signals for MCUs are relatively fast compared to how quickly you can press and release a hardware actuator device such as a button. This brings up two serious issues that the system designer must deal with in order ensure the overall circuitry (both hardware and firmware) will work properly under all possible conditions. The two issues are: 1) "noise" on the interrupt signal, and 2) the duration of the pulse physically connected to the interrupt input on the MCU.

Noise on the signal can come in many forms, but we'll only deal with two forms in this discussion. For the first form of noise, assume the signal connected to the MCU is the output of another electronic (non-mechanical) device. Under these circumstances, we'll assume that this device is smart enough of configured such that it can notify the MCU of an interrupt without causing other issues (meaning the pulse-width of the signal is short enough not to cause problems when the MCU processes the interrupt, a topic we'll discuss later). Any signal such as this could be affected by general noise components in the system (EMF, transmission life effects, etc.). This type of noise could be a pulse that could effectively generate an interrupt on the MCU.

For the second type of noise, assume the press of a button generates the signal connected to the MCU. In this case, you need to deal compensate for the issues of "switch bounce". Every mechanical actuator device has bounce issues. This means that if you press the switch once, the switch contacts can actually "bounce" a few times before arriving at a steady state value. The result is that a single button press can generate a separate pulse from each switch bounch. Because the contacts can bounce for up to 50ms, each of the bounces can generate a separate interrupt. Keep in mind that the MCU executes instructions on the nano-second level, in which case, a micro-second is plenty of time for the MCU to deal with the interrupt.

The solution for these types of noise is to apply a "debouncer" to the switch outputs. All switches must be debounced; you have the choise of debouncing them in hardware of firmware. The debounce circuit works by first detecting a change in the signal value, waiting for a specific amount of time, then checking the signal value again. If the signal value indicates the signal is "off", then the signal must have been noise, so the debouncer does not pass the signal to the MCU. On the other hand, if the signal is still in its "on" state, the debouncer passes the "on" value of the signal to the MCU, which then processes the interrupt.

There is another issue you need to deal with in addition to debouncing the circuit; this issue has to do with the width of the pulse on the signal connected to the MCU interrupt. In actuality, chances are good that when you press the button, the button will remain pressed long enough for the RAT MCU to process the interrupt and return to normal program execution. This presents the situation that when the MCU exits the interrupt service routine and the interrupts are unmasked, the interrupt from your last button press will still be there because you have not yet lifted your finger. In this situation, the hardware notices that the interrupt line is still asserted and enters back into a second round of interrupt processing for the same interrupt (namely the initial single button

press). In effect, the MCU would service the same interrupt multiple times, which probably is not what you want.

To avoid the situation where a single "event" can generate multiple interrupts, you can connect the signal that would that indicates a device needs attention to a "mono-stable multivibrator", commonly known as a "one-shot". As the name implies, this device has one stable state, and one non-stable state, or temporary state. The "on" state is the non-stable state, which means it's only temporarily in that state. The stable state is the off state. When you connect a button to a one shot, the output of the one-shot is only asserted for a fixed length of time, which officially makes it independent of the length of time the button is pressed for. The input of the one-shot connects to the output of the device generating the interrupt signal; the output of the one-shot connects to the RAT MCU's interrupt input. The one-shot circuitry thus provides a relatively short pulse output to the MCU input; this pulse is short enough to ensure that the MCU will only process one interrupt per button press.

Figure 18-24 shows a diagram outlining the debouncing and one-shot issues. The signal labeled (A) represents the signal from the button. This signal shows a button press and the actual reaction on the signal due to switch bounce. The signal labeled (B) represents the ideal output of the button press. Note differences between signal (A) and (B) are the toggling of the switch after the initial actuation. The signal labeled (C) represents the classic debounced button, which shows the switch actuates only after the switch has complete it bounce routine. The debounced characteristic in (C) is fine for some applications, but not for signals that connect directly to the RAT interrupt input. The signal in (D) shows what the RAT MCU requires, which is essentially a signal that is both debounced and connected to the a one-shot.

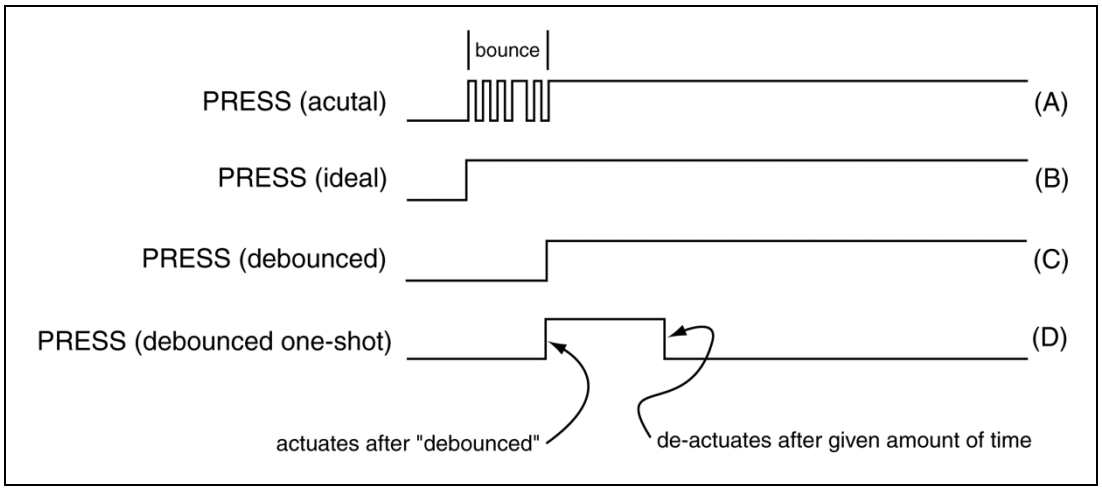


Figure 18-24: Example oscilloscope output for debounce problem.

The pulse-width of the debounce circuit's output acts independently of the input to the one-shot. In this way, the input signal can remain high for an indefinite period while the output signal remains only briefly remains asserted before it returns to zero. The high state of the debounce circuit's output is referred to as the unstable state (because it's momentary) while the low state is the stable state. There is only one stable state, hence, the circuit exhibits mono-stability. The debounce circuit synchronizes with the MCU's system clock. The final result is that the pulse is long enough to cause the RAT MCU to go into an interrupt cycle (meaning the pulse will be present at the end of the execute cycle) and that the pulse will be not so long that it gone before the end of the next execute cycle. In particular, these issues are:

1. If the interrupt pulse is too short, the RAT MCU may not recognize it and the interrupt will effectively go away without the MCU entering into an interrupt cycle. Making the one-shot pulse two clock cycles and synchronizing those clock pulses to the systems clock ensures that the RAT MCU will see an interrupt but not act more than once on that interrupt.

2. If the interrupt connects to a button and the button is bouncing, the MCU may attempt to process more interrupts than what actually arrived. While having the MCU automatically mask the interrupts solves this problem, the interrupt can still be processed fast enough in one "bounce pulse" such that the MCU will act on another bounce pulse, which is rarely what you're intending to do.

Figure 18-25 shows timing diagram that describes the pertinent signal associated with an interrupt cycle. Here are some of the important items in Figure 18-25.

- The interrupt input in Figure 18-25 is the output of a one-shot, where the one-shot filters the INT input to the RAT MCU. For this example, there is no debounce connected to the input. Specifically, the one-shot guarantees the signal being input to the INT input on the RAT MCU is not too long or too short, which are cases that could potentially be problematic for the RAT MCU. The one-shot in the diagram is asserted as short as it possibly can be while still assuring the signal will generate an interrupt on the RAT MCU. Recall that the INT signal must be asserted on the INT input at the end of the control unit's execute cycle.

- The EXT_INT signal connects to a one-shot; the output of the one-shot connects to the INT input of the RAT MCU. Note that even though the EXT_INT signal remains asserted, the INT signal remains two clock periods in length due to the filtering effects of the one-shot.

- The value in the PC is arbitrary and the EXT_INT signal asserts at an arbitrary time. Recall that the interrupt signal is an output from external hardware; it is therfore unlikely that any external hardware is synchronized the RAT MCU's system clock.

- Entering into the interrupt cycle causes the RAT MCU hardware to load 0x3FF, the interrupt vector address, into the program counter. The instruction at 0x3FF is an unconditional branch that causes the hardware to load the address of the interrupt service routine into the PC, which is 0x07C for this example. Normal instruction execution resumes at that point (meaning sequential instruction execution).

- For this example, we re-enable the interrupts as soon as possible. Recall that RAT MCU interrupt architecture automatically masks the interrupts when it enters an interrupt cycle. For the example in Figure 18-25, the first instruction after the execution of the BRN instruction (the instruction at the interrupt vector address, or program memory address 0x3FF) is an SEI instruction. Note that the SEI instruction enables the interrupt after it completes the execute cycle associated with the SEI instruction. The underlying point of this example is that the INT signal must be at least two clock cycles in length to be sure that the MCU acts on it, but should not be more than six clock cycles or else it may generate another interrupt in a special case such as this.

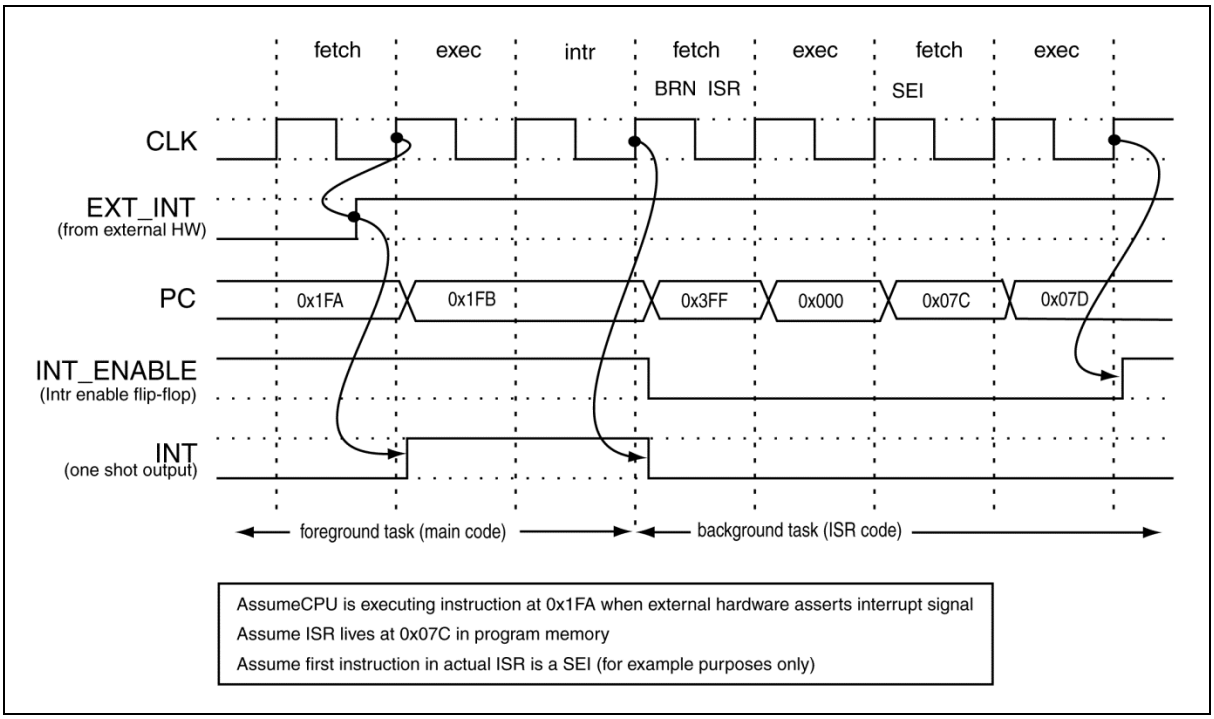


Figure 18-25: A timing diagram showing an interrupt processing.

18.8.3 Interrupt Support Hardware

Two simple pieces of hardware support the RAT interrupt architecture: the interrupt shadow registers and the interrupt enable circuitry.

18.8.3.1 The Interrupt Shadow Registers

The interrupt shadow registers, or simply “shadow registers”, form the heart of the context savings mechanism for the RAT MCU. The context savings in the RAT MCU comprises of saving the value of the C and Z flags when the RAT MCU acts on an interrupt. The RAT MCU implements both the normal C and Z flag as D flip-flops (with features); therefore, it should be no surprise that the RAT MCU also implements the shadow C and the shadow Z flags as D flip-flops.

There are two main aspects to the interrupt shadow registers: the context saving part and the context restoration part. The context savings part requires that the normal C and Z flag contents be saved in the shadow C and Z flags when the RAT MCU goes into an interrupt cycle. This requires a path from the output of the normal C and Z flags to the input of the shadow C and Z flags, respectively. Both the normal and shadow C and Z flags have load controls inputs to allow the loading of the flags at the appropriate times. Neither the normal or shadow C and Z flags are under direct program control; loading of these flags only occurs with the appropriate ALU-based instruction for the normal flags and as part of an interrupt cycle for the shadow C and Z flags.

The context restoration mechanism comprises of copying the contents of the shadow C and Z flags into the normal C and Z flag. The context savings mechanism requires data path routing from the output of the shadow C and Z flag to the input of the normal C and Z flags. While the inputs to the shadow registers only have one input (from the normal flags), the normal C and Z flags can be loaded from either the shadow registers or from the C and Z outputs of the ALU. The loading of the normal flags is thus an output of a 2:1 MUX. The control unit provides all the control signals (loading and MUX controls) for both the shadow and normal registers. Once again, the RAT MCU instructions implicitly control of both sets of condition registers, meaning there is no

instruction that directly controls these registers. The context restoration mechanism, and thus the loading of the shadow registers into the normal registers occur as a side effect of the RETID and RETIE instructions.

Figure 18-26 a detailed drawing of the complete FLAGS module. This module supports the loading of the C and Z flags from the ALU (for ALU-type instructions) or from the shadow flags (for the context restoration associated with the RETID and RETIE instructions). There are a few items worthy of noting from this diagram.

- The shadow C and Z flags effectively share two signals: FLG_SHAD_LD and FLG_LD_SEL. The FLG_SHAD_LD signal controls the context saving while the FLG_LD_SEL controls the context restoration.

- The FLG_LD_SEL signal uses a shorthand notation to show that both MUXes share the same signal. This is common schematic notation so do not allow it to confuse you.

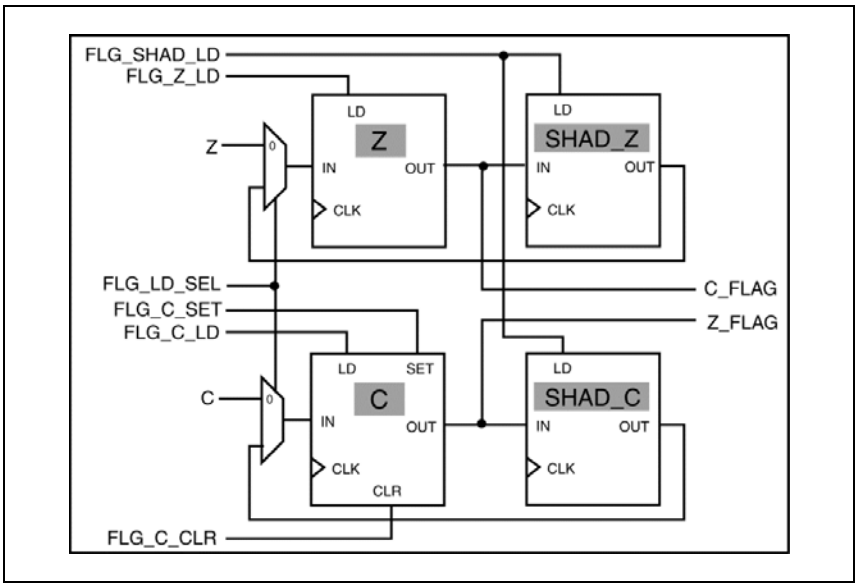


Figure 18-26: A diagram showing the complete set of C and Z registers.

18.8.3.2 The Interrupt Masking Circuitry

The interrupt masking circuitry allows for direct masking and unmasking of the RAT MCU's interrupt under both hardware and program control. The programmer uses the SEI and CLI instructions to unmask (enable) and mask (disable) the RAT MCU interrupts, respectively. The heart of the interrupt masking circuitry is a flip-flop; the hardware ANDs this flip-flop's output with the external interrupt signal. The output of this AND gate connects to the interrupt input on the RAT MCU.

The hardware designer can implement the flip-flop itself in several different ways. Probably the most straight-forward approach is to model the flip-flop with no data input, but have a synchronous input that sets the flip-flop for unmasking (thus enabling the AND gate) or clears the flip-flop for interrupt masking (kills the AND gate[10]). Taking this approach requires that the interrupt masking circuitry have two control signals: a set input and a clear input. Anyway you approach this circuitry requires two inputs. Another approach would be to have a data input and a load input; for masking the interrupts, you would set the data input to '0' and then load the flip-flop (with the converse for unmasking the interrupts.

[10] The notion of killing an AND gates means you force one input to logical '0', which ensures the output of the gate will always be '0'.

18.8.4 Interrupts and Program Flow Control

The notion of program flow control in interrupt processing is similar to the program flow control associated with subroutine processing. The processing of the instructions in the interrupt service routine represents normal instruction processing with no new details. What we are interested are the steps the hardware takes upon acknowledging interrupts and returning from interrupts. The following sections describe those details as they relate to the underlying RAT MCU architecture.

18.8.4.1 Acting on Interrupts

Normal program flow control is "interrupted" (for lack of a better word) when two conditions happen: 1) when the interrupts are unmasked (enabled), and 2) then the value on the interrupt input pin on the RAT MCU is at a logical '1' value.

When the signal on the interrupt input is asserted and interrupts are unmasked, interrupt processing begins. The interrupt signal must be a pulse of a particular length in order for processing to continue. If the pulse is too long, the RAT will continue to see that the interrupt input is asserted and control will be passed immediately back to the ISR once ISR processing is complete. The interrupt signal needs to become unasserted in order for proper interrupt processing to occur. On the other hand, if the interrupt pulse is too short, the RAT MCU may not recognize it and that the signal on the INT input is asserted and the interrupt is effectively ignored. The shortest pulse used as an in interrupt signal for the RAT MCU needs to be a minimum length of two clock cycles to ensure proper acknowledgement.

When the RAT MCU goes into an interrupt cycle, the following things happen. After the RAT MCU implements the following bullets, official processing of the interrupt begins, as we consider operations before this step interrupt processing overhead. The RAT MCU's control unit is responsible for issuing the control signals that implement the following bulleted operations.

- The RAT MCU completes the instruction that it is currently executing. Keep in mind that the signal connected to the INT input on the RAT MCU can change asynchronously in relation to the RAT MCU's system clock. This means interrupts can occur during any phase of the system clock or instruction cycle. Before the handling of the interrupt begins, the execution of the current instruction completes. This RAT MCU's control unit assures that this functionality occurs.

- The RAT MCU transfers program control to a pre-determined address in instruction memory. In the case of the RAT MCU, the address is 0x3FF. We refer to this address as the *interrupt vector address.* The RAT MCU loads this value into the program counter as part of the interrupt cycle. Recalling the interrupts are similar to subroutines, the RAT MCU's hardware pushes the address of the next instruction that would have been executed had the RAT MCU not entered the interrupt cycle onto the stack. The RAT MCU pops this address off the stack when it completes execution of the interrupt service routine.

- It is the programmer's responsibility to place an unconditional branch instruction (BRN) at the vector address such that program control transfers to the interrupt service routine after the MCU executes this instruction. Upon leaving the interrupt cycle (a state in the control unit's FSM) and entering normal processing, the RAT MCU executes the BRN instruction. The precise location of the ISR is not important, however, the ISR code should not be reachable from any other code in the program. If this were the case, the special ISR return instruction will cause spastic behavior in the main program; spastic programs are undesireable. The code in the ISR implements some task that appropriately services the interrupt.

- The RAT MCU hardware saves the current operating context. This step comprises of copying the state of the normal C and Z flags to the shadow C and Z flags. The control unit issues the appropriate control signals for the context saving during the interrupt cycle.

- The RAT MCU automatically masks the interrupt by clearing the interrupt flip-flop. The control unit is responsible for this action as part of the interrupt cycle. The hardware does this step; keep in mind that the hardware does not issue a CLI instruction.

18.8.4.2 Returning From Interrupt Processing

When the ISR is complete, the RAT MCU returns program control to the instruction after the instruction that was executing when the interrupts cycle began. The complete sequence of events is as follows:

- The program alerts the RAT MCU hardware to the fact that it has completed processing of the interrupt service routine. The program does this by issuing a special return from interrupt instruction. There are two "return from interrupt" instructions in the RAT MCU instruction set: RETIE and RETID. These two instructions differ by the notion that they either unmask or mask the interrupts as part of the instruction. This control of the interrupt masking is part of the two instructions; the control unit issues the appropriate control signals for these actions. Recall that the RAT MCU automatically disabled the interrupts upon entering the interrupt cycle; note that the programmer has a choice as to whether to leave the interrupts masked or unmasked upon completing interrupt processing.

- The RAT MCU restores the context when it executes an RETIE or RETID instruction. This means that the control unit sends out the appropriate control signals that cause the values in the shadow C and Z flags to load into the normal C and Z flags.

- The RETIE and RETID instructions are similar to the RET instruction in terms of stack operations. The difference between these instructions is that the RET instruction does not restore context. Executing the RETIE and RETID instructions does cause the RAT MCU to pop an address off the stack and place it into the program counter, using the exact same mechanism as the RET instruction. The control unit sends of the appropriate control signals to drive the current top of the stack to the PC and and increments the stack pointer.

18.8.4.3 Interrupt Architecture Summary

As you can see from the previous sections, the interrupt architecture of the RAT MCU entails a definite sequence of steps. This sequence of steps ensures a smooth transition to and from the interrupt service routine, as well as protecting the pre-interrupt operating context of the RAT MCU. Here is a brief summary of the steps involved with the acting on an interrupt, executing the interrupt service routine, and returning to the regularly scheduled processing.

- The RAT MCU currently has the interrupt unmasked and it detects an asserted signal on the interrupt input (INT)
- The RAT MCU completes execution of the instruction currently being executing
- The RAT MCU pushes the address of next instruction following the one last executed onto the stack
- The RAT MCU loads 0x3FF into the program counter
- The RAT MCU saves the present state of the condition flags to the shadow C & Z registers
- The RAT MCU executes the instruction at location 0x3FF (an unconditional branch instruction that directs program control to the ISR)
- The RAT MCU begins execution of the ISR
- The RAT MCU ends execution of the ISR
- The RAT MCU executes a RETIE or RETID instruction
- The RAT MCU pops the return address off the stack
- The RAT MCU copies the state of the shadow C & Z flags to the C & Z flags
- The RAT MCU executes the instruction following the one that was executing when the interrupt was received

18.9 Chapter Summary

- The control unit synchronizes all operations in the RAT MCU. The RAT MCU is a FSM that has condition flags, opcodes, and the interrupt as inputs, and the control signals to the various RAT MCU submodules as outputs. Execution of RAT MCU instructions is done during an instruction cycle, which comprises of a fetch and execute cycle. The fetch and execute cycles require one full clock cycle for execution as they represent the two states associated with normal program execution.

- The program counter generally points to the current instruction being executed. The program counter is a synchronously loadable 10-bit up-counter. This 10-bit counter provides the address range to index into the 1024x18 program memory.

- The register file, ALU, and condition flags handle the bulk of the RAT MCU's bit-crunching operations. The register file is a synchronous 32x8 dual-port RAM that is able to read two register file locations and write one register file location. The ALU can perform one of many data processing operations as instructed by the control unit. The condition flags are a C and Z flag implemented as flip-flops that provide information regarding the most recent results of the ALU.

- The stack pointer is a register that the RAT MCU uses to store the top-of-the-stack. The stack is inherently part of the scratch RAM so the stack pointer has an 8-bit width. The stack uses the direct output of the stack pointer for pop-type operations (pops, return from subroutines and interrupts) and one less than the current stack pointer value for push-type operations (pushes, subroutines calls and interrupt processing).

- The Input/Output architecture refers to the methods the RAT MCU handles basic communications with the outside world. The RAT MCU uses programmed memory for I/O. The interrupt input is part of the RAT MCU's I/O.

- Program flow control refers to any program flow that is not sequential. The possibility of non-sequential operation is associated with branch instruction, subroutine calls, and interrupt processing. The RAT MCU's hardware is responsible for ensuring the program flow control is correct for each of these instructions/operations.

- The interrupt architecture is a term we use to describe all the hardware and hardware-induced operations associated with the processing interrupts. The interrupt architecture is one of the first things you should examine when dealing with a new MCU or CPU, as interrupt driven programs have many distinct advantages over programs that are not interrupt driven.

18.10 Chapter Exercises

1. The C flag and Z flag are not equivalent (they have a different number of control inputs) in the RAT MCU. Briefly describe why they are not equivalent.

2. The C flag is an input to the ALU but the Z flag is not. Briefly describe why only the C input is fed back to the ALU.

3. Briefly describe why the RAT MCU has program control of the C flag (meaning it can set it and clear it using RAT instructions) but does not have the same control of the shadow C flag.

4. In the RAT MCU architecture, can you use the ST instruction to write a value to the stack? Briefly but complete explain your answer.

5. Briefly describe why the width of the scratch RAM is ten bits while the register data it stores is only eight bits.

6. Briefly describe the difference between an RET, a RETIE, and a RETID instruction.

7. The RAT MCU models the register file as a “dual-port RAM”. Explain what exactly is meant by a “dual-port RAM” in the context of how the RAT uses this device. Also, explain why it was necessary to use a dual-port RAM rather than a simpler memory device.

19 Miscellaneous RAT MCU Architecture Details

19.1 Introduction

While the previous chapters discussed many aspects of the RAT MCU, there are a few more topics we need to introduce to provide you with the overall big picture. When we say "big picture", we mean the big picture in terms of both the RAT MCU and basic computer architectures in general. In truth, we could not easily introduce a few subjects out there earlier because we had not yet provided you with the background to facilitate that discussion. This chapter hopefully ties together many of the issues regarding the RAT MCU and computer architecture in general.

Main Chapter Topics

- **OVERVIEW OF IMPORTANT RAT MCU TIMING ISSUES:** This chapter describes the various important timing issues as they apply to executing RAT MCU instructions. In-depth description of timing diagrams support this discussion.
- **OVERVIEW OF THE RAT MCU WRAPPER:** This chapter describes the "wrapper" which we use to interface the RAT MCU with external hardware such as a development board or other modules.
- **OVERVIEW OF RISC AND CISC ARCHITECTURES:** This chapter describes the RISC and CISC architectures including their main accepted differences.
- **OVERVIEW OF LEVELS OF MEMORY:** This chapter describes the notion of memory levels as they relate to basic computer systems.

Why This Chapter is Important

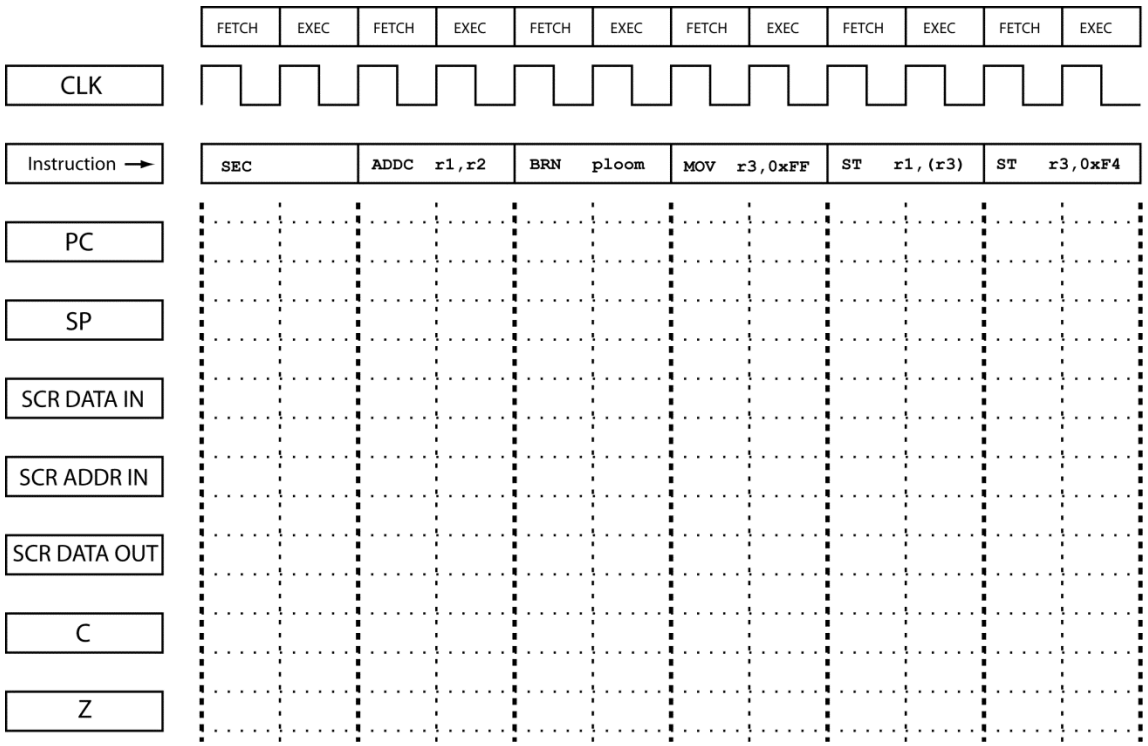
This chapter is important because it describes some the non-architectural but still important details involving the RAT MCU.

19.2 RAT MCU Timing Issues

You can't fully understand the lower-level operations of the RAT MCU unless you have a firm grasp on the underlying timing issues associated with the RAT MCU instruction set. This section outlines the underlying timing issues by solving a few key timing problems associated with RAT MCU instructions. The idea behind this section is to convince you that the operation of the RAT MCU is fully deterministic and relatively simple once you fully understand all aspects of the RAT MCU hardware and how the RAT MCU instructions interact with that hardware.

Example 19-1: RAT Instruction Timing Example 1

Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x0AA, SP = 0xFA, C = '0', and Z = '0'. Additionally, the "ploom" label has a value of 0x1DD, r1 holds a value of 0x4F, and r2 holds a value of 0xB0.



Solution: Figure 19-1 shows the solution to Example 19-1.

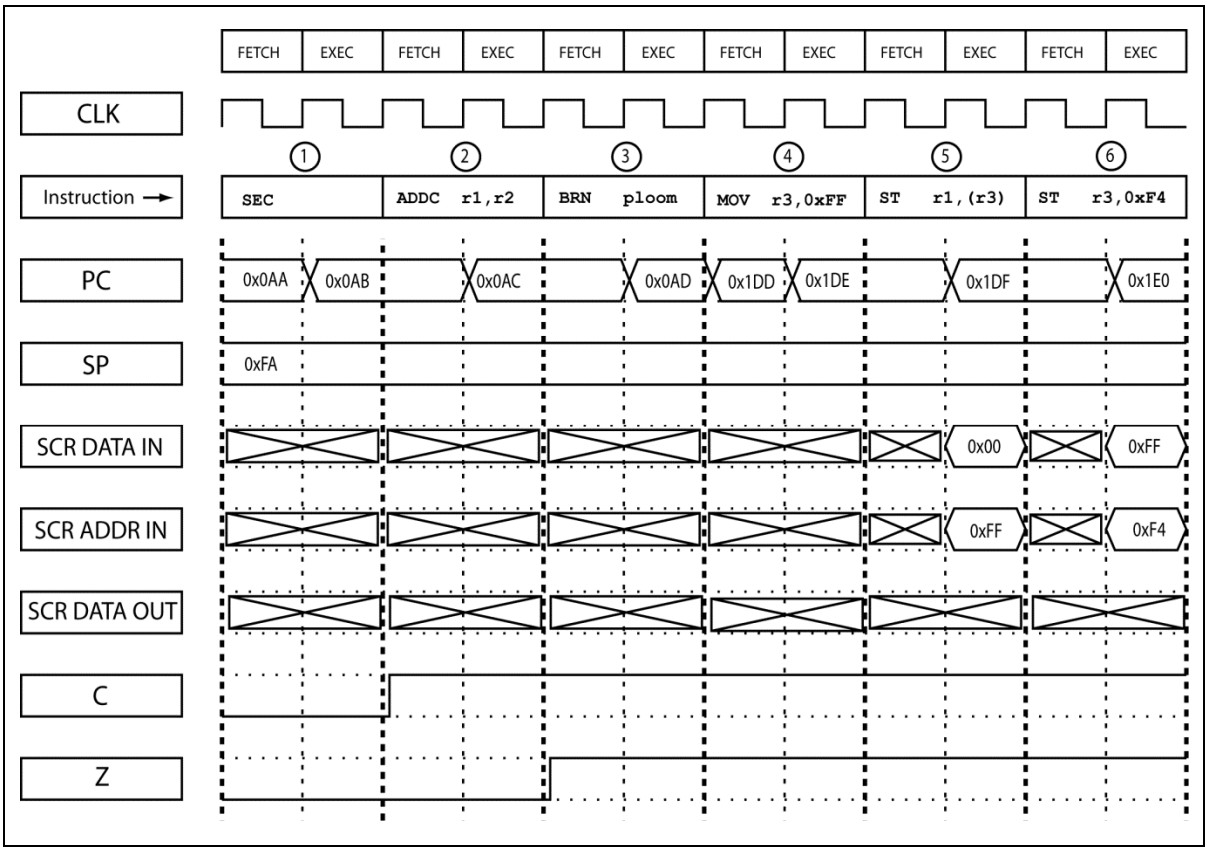


Figure 19-1: The solution to Example 19-1.

Here are some interesting items regarding the solution to Example 19-1. We'll discuss each instruction according to the numbers appearing above the solution in Error! Reference source not found..

(1) The start of this instruction contains all the initial values given in the problem description. This instruction is a SEC instruction, which set the carry flag. The PC increments after completion of the fetch cycle. This instruction has nothing to do with scratch RAM, so we've put a giant X in those locations. Keep in mind that there are actually input and output values on the scratch RAM, but they are incidental and meaningless to instructions that do not use them. The C flag sets during the following fetch cycle as a result of asserting the set signal on the C flag during the execution cycle associated with the SEC instruction.

(2) The ADDC instruction adds the two operands and the C flag, or in this case, 0x4F + 0xB0 + 0x01. The result of this addition writes back to the register file under control of the control unit. The result of this addition is 0x00 with a C of '1'. Additionally, the Z flag is set to '1' because the result is zero. This instruction does not use the scratch RAM.

(3) The unconditional branch instruction loads the address value associated with the "ploom" label into the PC. Recall that the assembler associates a prog_rom address value with the "ploom" label; this label represents the address of the instruction the program is branching to. Note that the PC increments to 0x0AD during the execution cycle of the BRN instruction, but never uses this value. What happens instead is that the PC loads the address associated with the instruction to branch to in the fetch cycle of the next instruction.

(4) The reg/immed form of the MOV instruction writes an immediate value to the register file. In particular, the value of 0xFF writes to register r3. The MOV instruction does not involve the

"DX_OUT" (OP_A) output of the register file and only uses the "DY_OUT" output (OP_B). This instruction does not use the scratch RAM.

(5) This instruction is the register indirect version of the ST instruction. This instruction writes the contents of register r1 to scratch RAM at the address provided by the value in register r3. The value in register r1 is the result of the previous ADDC instruction, which in this case if 0x00. Thus the scratch RAM address input has the value of r3 (0xFF) while the scratch RAM data input has the value of register r1 (0x00). These values are only guaranteed to be there after entry into the execute cycle of the instruction. The scratch RAM data output has something, but we simply don't care what it is as this is a store instruction.

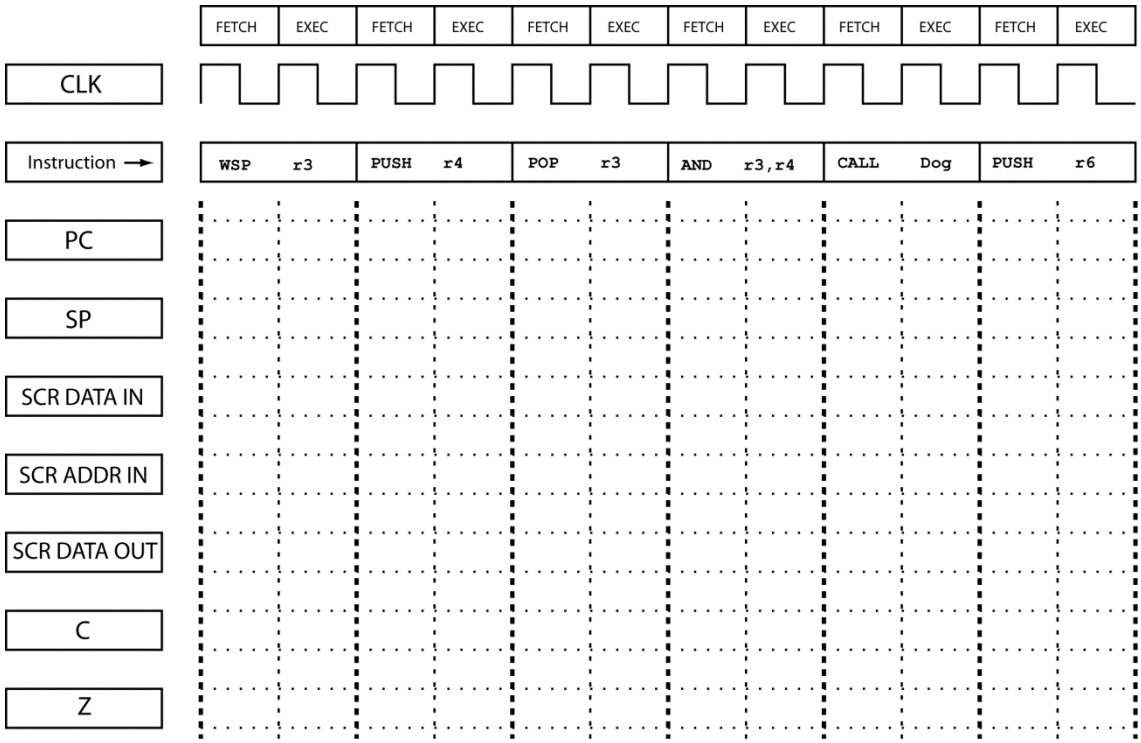
(6) This instruction is the reg/immed version of the ST instruction. This instruction writes the contents of register r3 to scratch RAM at the address given by the second operand of the instruction: 0xF4. The value in register r3 is the result of the previous MOV instruction, which in this case if 0xFF. Thus the scratch RAM address input has the value of r3 (0xF4) while the scratch RAM data input has the value of register r1 (0xFF). These values are only guaranteed to be there after entry into the execute cycle of the instruction.

Here are some other interesting and important items to note:

- No instruction accesses the stack pointer, so it never changes state.
- Only the SEC and ADDC instructions involved the condition flags.

Example 19-2: RAT Instruction Timing Example 2

Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x082, SP = 0xFF, C = '1', and Z = '1'. Additionally, the "Dog" label has a value of 0x112, r3 holds a value of 0xE8, r4 holds a value of 0x23, and r6 holds a value of 0xAB.



Solution: Figure 19-2 shows the solution to Example 19-2.

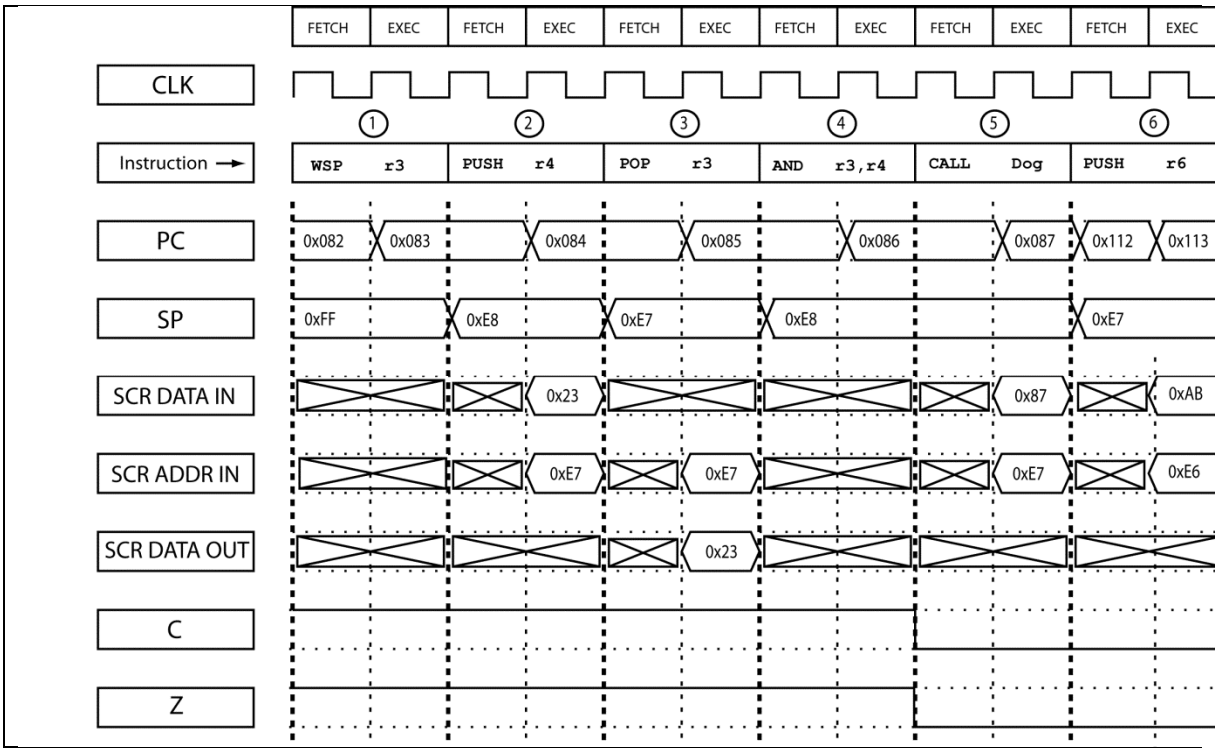


Figure 19-2: The solution to Example 19-2.

Here are some interesting items regarding the solution in Example 19-2. We'll discuss each instruction according to the numbers appearing above the solution in Figure 19-2**.**

(1) The WSP instruction writes a value from a register to the stack pointer. The stack pointer does in fact have an initial value, but the WSP instruction will overwrite that value. The register file drives the new value from a register to the stack pointer on the execution cycle of the WSP instruction; the actual writing to the stack pointer occurs at the beginning of fetch cycle for the following instruction. As with all instructions, the program counter increments at the beginning of the execute cycle for the current instruction.

(2) The PUSH instruction saves the value in register r4 onto the stack. The value in register r4 (0x23) appears on the scratch RAM's data input during the execution cycle of the PUSH instruction when the register file places its value on the DX_OUT. The PUSH instruction causes the stack pointer to decrement, which actually occurs during the beginning of the fetch cycle for the next instruction. The PUSH instruction chooses the "-1" version of the stack pointer and uses that value as the address input to the scratch RAM.

(3) The POP instruction transfers the value in the top of the stack to a register file register. In this case, the POP instruction loads the top of the stack to register r3. The POP instruction causes the stack pointer to increment, which officially occurs at the beginning of the fetch cycle for the next instruction. The value on the scratch RAM's data output is the value 0x23, which was the value written to the scratch RAM by the previous instruction.

(4) The AND instruction drives the contents of register r3 to the ALU. Both operands of this instruction contain the same non-zero value, which generates a non-zero result. The non-zero result has the effect of clearing the Z flag; the AND operation does not affect the C flag.

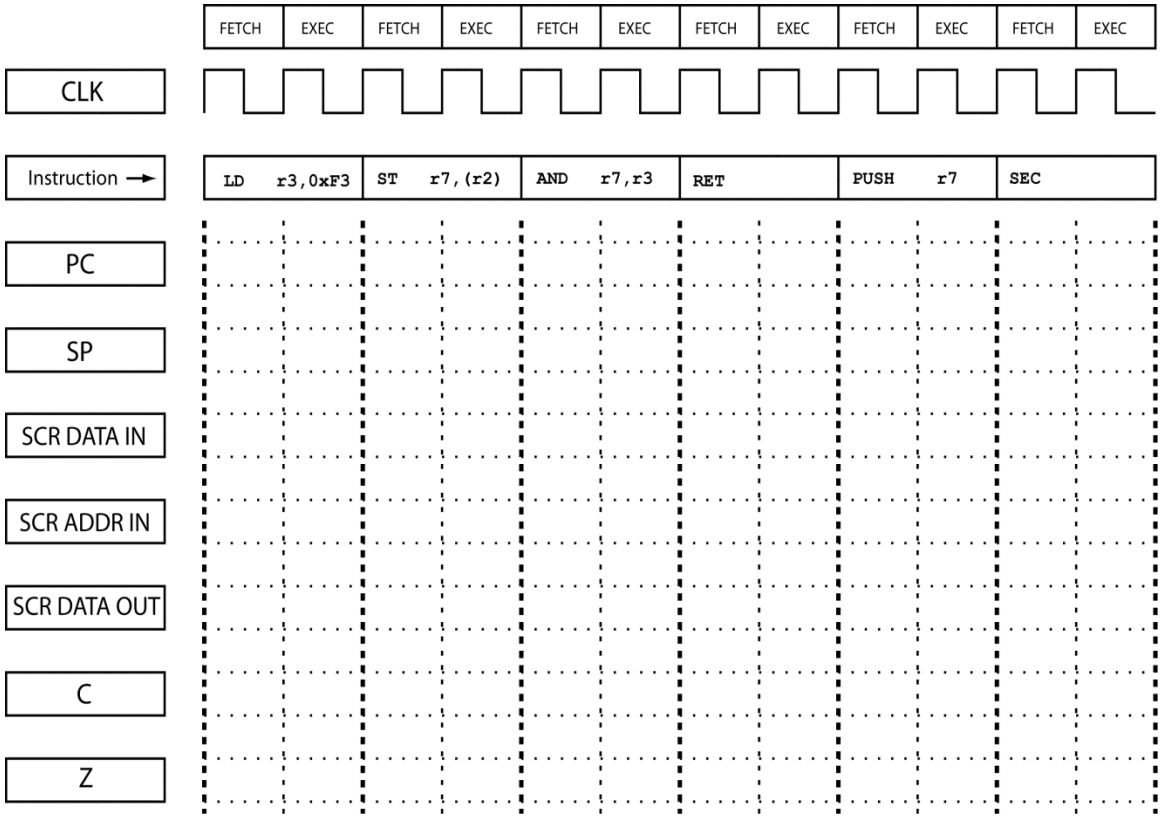
(5) The CALL instruction transfers program control to the address associated with the "Dog" label. This program control transfer happens when the value associated with the "Dog" label is loaded into the

program counter. Despite this being a CALL instruction, the program counter increments after the fetch cycle as it always does. However, the CALL instruction will then load the new value into the program counter at the beginning of the fetch cycle of the following instruction, which is the fetch cycle of the sixth instruction. Additionally, the CALL instruction causes a decrement of the stack pointer, which once again occurs at the beginning of the fetch cycle in the following instruction. The CALL instruction is similar to a PUSH instruction, so the scratch RAM's address input is one less than the current value in the stack pointer. The value in this case is 0x87.

(6) The final instruction is another PUSH instruction. This instruction causes the stack pointer to decrement, but since the decrement occurs at the beginning of the next fetch cycle, you don't see it in this timing diagram. The PUSH instruction saves r6 onto the stack, which requires the register file to drive the contents of register r6 to the scratch RAM's data input. The scratch RAM's address input is one less than the current stack pointer value, as the PUSH instruction selects the "-1" input as the address input to the scratch RAM.

Example 19-3: RAT Instruction Timing Example 3

Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x200, SP = 0xE4, C = '0', and Z = '0'. Additionally, the top of the stack points at the value of 0x015, the scratch RAM address of 0xF3 holds a value of 0x055, r7 holds a value of 0xAA, and r2 holds the value 0x67.



Solution: Figure 19-3 shows the solution to Example 19-3.

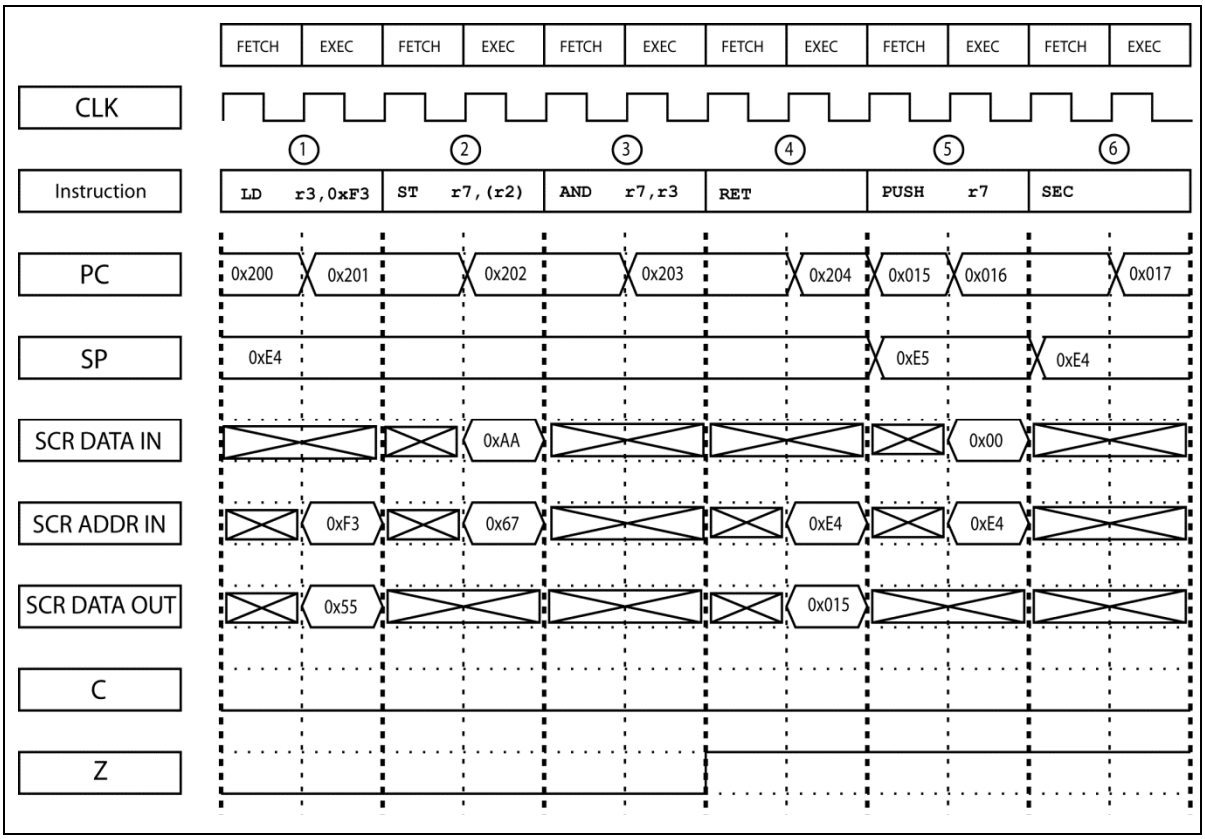


Figure 19-3: The solution to Example 19-3.

Here are some interesting items regarding the solution in Example 19-3. We'll discuss each instruction according to the numbers appearing above the solution in Example 19-3**.**

(1) The first instruction is a load immediate type instruction that loads data from the scratch RAM at address 0xF3 into register r3. The problem states that scratch RAM address 0xF3 holds the value of 0x55, which causes that value to be sent to the register file from the scratch RAM. The scratch RAM data input for a LD instruction has no use so we mark it with the big ugly X thang. Similarly, scratch RAM's address input and data output during the fetch cycle of this instruction also have no meaning. There is data on those lines, but the data has no meaning. The program counter increments at the beginning of the fetch cycle for this instruction, as it always does.

(2) The ST instruction copies the value in the register file to the scratch RAM at the address provided in register r2. This is the indirect version of the ST instruction. The problem statement gives the value in as register r7 as 0xAA, which is the value routed from the DX_OUT output of the register file to the scratch RAM. The scratch RAM's data output has no meaning for this instruction. The scratch RAM's address input and data input also have no meaning during the fetch cycle of this instruction.

(3) The AND instruction is the first instruction in this problem that affects either the C or Z flag. This instruction ANDs the contents of registers r3 and r7 with the result being stored in register r7. The value in register r3 is 0x55 (resulting from the LD instruction) while the value in register r7 is given as 0xAA. The result of the AND instruction is 0x00, which causes the Z flag to set starting at the beginning of the fetch cycle for the next instruction.

(4) The RET instruction represents a return from a subroutine. The RET instruction pops the return address off the stack in a normal pop-type action. The problem cleverly states that the top of the stack points to a value of 0x015, which is the value that the instruction places in the program counter at the beginning

of the fetch cycle of the next instruction. The popping action of the stack causes the stack pointer to increment, which again happens as a result of the control signal that are set as part of the RET instruction; the actual writing of the stack pointer occurs at the beginning of the fetch cycle in the next instruction. The address copied from the scratch RAM to the PC is at scratch RAM address pointed to by the current SP value, which is 0xE4; this data is available during the execute cycle of the instruction. The data on the scratch RAM's output is the address data that will become the new program counter value. The program counter increments normally for this instruction to 0x204, but the instructions never actually use this value as the correct program counter value lives at the addresses pointed to by the stack pointer (the top of the stack).

(5) The PUSH instruction stores the value in register r7 on the stack at one less than the current address in the stack pointer. The value in r7 is 0x00 resulting from the previous AND instruction; the register file routes this to the scratch RAM's data input as part of the PUSH instruction. The PUSH instruction causes the stack pointer to decrement, but the actual decrement does not appear on the stack pointer until the beginning of the fetch cycle of the next instruction; it of course appears as if the stack pointer was decremented for this instruction as the scratch RAM's data input shows it's been decremented in the execute cycle, but this is because it is choosing the "-1" input, with the "-1" being relative to the current stack pointer value.

(6) The final instruction is a SEC, which sets the C flag. No instruction in this sequence affects the C flag except for this instruction. The key to this instruction is realizing that the C flag does not actually change state until the beginning of the fetch cycle of the next instruction. However, since this is the last instruction in this example problem, the diagram does not show the change. Recall that the change in the stack pointer for this instruction happens because of the previous instruction.

19.3 The RAT MCU Wrapper

I'm not actually sure where the term "wrapper" came from. Someone, or maybe I, was working with may have made it up for all I can remember. Despite its dubious origins, the notion of a wrapper is rather important in the land of softcore MCUs. The sole purpose of the wrapper is to provide an interface between the RAT MCU and the development board you implement it on. As you'll see, there are issues that the RAT MCU simply does not handle, and it thus relies on the wrapper for support.

The main purpose of a development board is to provide a set of input/output "options" for whatever you're implementing on the board. We're implementing a microcontroller, so we'll discuss three types of I/O: 1) an input device (such as a button), 2) an output device (such as an LED), and 3) a generic pin (meaning you can assign the pin as either being an input or an output). For this section, we'll only consider the first two types of I/O and we'll base that I/O on the black box diagram of Figure 19-4.

Figure 19-4 shows the black box diagram of a development board containing 12 inputs and 20 outputs. The inputs are a set of four buttons and a set of eight switches. The outputs are a set of eight LEDs, a set of eight segments, and a set of four anodes. The segments and anodes support a four-digit common anode seven-segment (including decimal point) display.

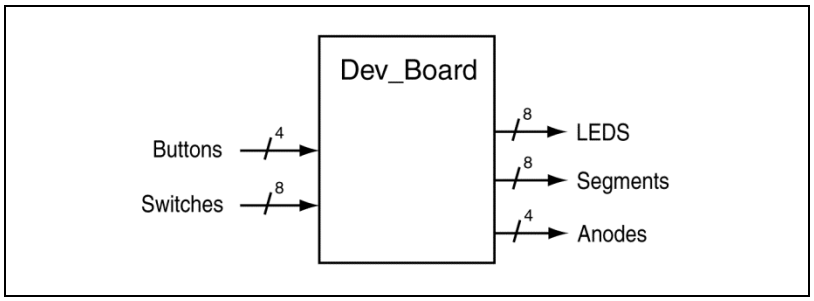


Figure 19-4: An example of a development board that we'll discuss in this section.

The lower-level details of the wrapper in the context of the RAT MCU involve how to interface the IN and OUT instructions with the wrapper. In other words, we must design the wrapper in such a way as to make the IN and OUT instructions function properly. Figure 19-5 shows a high-level model of the hardware that highlights the input and output hardware, which is what we're interested in for this discussion.

We implemented the RAT Wrapper as a simple VHDL model; you can find the full model in the appendix. The approach we'll take here is to describe the more important parts of the wrapper model by examining the VHDL model and relating that model to real world digital parts that you know and love. The items of interest in Figure 19-5 are the shaded boxes, as they represent the only hardware other than the RAT MCU itself.

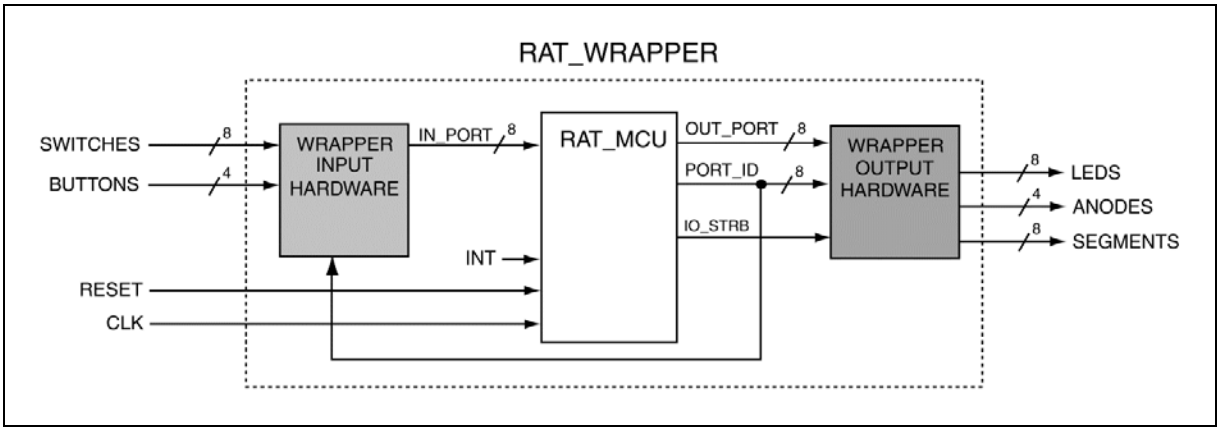


Figure 19-5: The RAT wrapper showing the RAT MCU and I/O support hardware.

The big question that I always ask people (and they generally do not know or do not say) is what is the main difference between the statements in Figure 19-6 and Figure 19-7? The big hint for this question is in regards to the basic hardware that the VHDL models induce. The answer is that the code for the input wrapper hardware in Figure 19-6 does not induce memory while the code in Figure 19-7 does. This difference is significant for the proper operation of the RAT MCU on a development boards.

Examining Figure 19-6 closely should remind you that the code models a simple MUX. The model chooses one of the input vectors to transfer to the input port of the RAT MCU. The port ID acts as the select inputs to the RAT MCU. In other words, when the select inputs match the preset value for the given input, the MUX passes that data through to the RAT MCU; otherwise, it passes 0x00 to the MCU. Maybe a better way to word this would be when the port ID matches one of the preset 8-bit values, the data passes through the MUX to the RAT MCU. In the cases where there is a match, the IN instruction is expecting data from the port ID associated with the IN instruction (the right operand); it places that data in the register also associated with the IN instruction (the left operand).

The code in Figure 19-6 does not show that the "s_port_id" (which connects to the PORT_ID output of the RAT MCU) is associated with the IN instruction while the "s_input_port" signal (which connects to the IN_PORT input of the RAT MCU) is associated with the actual data input. One other nice feature regarding the code in Figure 19-6 is that is uses the VHDL "constant" feature to associated the port ID number with a label as a self-commenting and easy code modification feature.

```
-------------------------------------------------------------------------
-- INPUT PORT IDS -------------------------------------------------------
CONSTANT SWITCHES_ID : STD_LOGIC_VECTOR (7 downto 0) := X"20";
CONSTANT BUTTONS_ID  : STD_LOGIC_VECTOR (3 downto 0) := X"24";
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- MUX for selecting what input to pass to RAT MCU
-------------------------------------------------------------------------
inputs: process(s_port_id, SWITCHES)
begin
   if (s_port_id = SWITCHES_ID) then
      s_input_port <= SWITCHES;
   elsif (s_port_id = BUTTONS_ID) then
      s_input_port <= BUTTONS;
   else
      s_input_port <= x"00";
   end if;
end process inputs;
-------------------------------------------------------------------------
```

Figure 19-6: VHDL model for the "wrapper input hardware" module Figure 19-5.

Figure 19-7 shows the VHDL source code for the "wrapper output hardware" box in Figure 19-5. The first thing you should notice about the code in Figure 19-7 is that the if statement's lack an else, which of course means that this source code induces memory. This fact is extremely important regarding the output devices on the development board because the output devices need to hold their state. For example, when your code outputs a value to the LEDs such as a value that turns the LEDs on, you expect the LEDs to remain on after the RAT MCU completes execution of the instruction. This persistence does not come automatically; the truth is that you must register each output device in order to give the devices a state. Recall that a "state" is memory, and recall that a register is an edge-sensitive memory device.

The code in Figure 19-7 has some interesting features that you must understand; here is an overview of those features.

- As you can see from Figure 19-7, the RAT MCU wrapper support three types of output devices: eight LEDs, eight segments, and four anodes. This means that the model in Figure 19-7 contains three registers: two eight-bit registers for the LEDs and segments and one four-bit register for the anodes.

- The source code for the wrapper output hardware synchronizes the latching of output data to the output registers with the system clock and the IO_STRB signal. As you can see from the entire VHDL wrapper model (find it in the appendix), the "s_load" signal in Figure 19-7connects to the IO_STRB signal, which is an output of the RAT MCU. Recall that the IO_STRB signal is a one system clock-cycle pulse that is output as a result of the RAT MCU executing an OUT instruction.

- The OUT instruction has two operands: the register whose data is being output (the left operand) and the port ID of the output device that the output data will be latched into. The code in Figure 19-7 "decodes" the input port ID, which allows it to send the load signal to the appropriate output register. The key here is to make sure the port IDs you use in your code match to the port IDs that the wrapper model is using[1].

[1] This applies to the input wrapper hardware also.

- The code in Figure 19-7 essentially models a decoder and three registers. The input to the decoder is "s_port_id" signal, which connects to the PORT_ID signal on the RAT MCU. The decoder has three outputs, which serve as the load enable signals to the three output registers.

- The code in Figure 19-7 uses the VHDL constant keyword to define the port IDs for the given output hardware devices. This allows for quick code modifications if the port IDs were to ever change. Keep in mind that the choice of port IDs for in this example is completely arbitrary; in real life, you may have reasons to assign them more robustly.

- For your viewing pleasure, Figure 19-8 provides the probable hardware configuration induced by the VHDL source code in Figure 19-7. Note that all the signal names are consistent throughout this discussion.

- The source code uses an "r_" prefix for the labels associated with the labels associated with the registers. Using this prefix alerts the astute hardware engineer that the label represents storage, namely a register. This is good VHDL coding practice, which is similar to using an "s_" prefix for signals and a "v_" prefix for variables.

```
-------------------------------------------------------------------------------
-- OUTPUT PORT IDS ------------------------------------------------------------
CONSTANT LEDS_ID       : STD_LOGIC_VECTOR (7 downto 0) := X"40";
CONSTANT SEGMENTS_ID   : STD_LOGIC_VECTOR (7 downto 0) := X"82";
CONSTANT DISP_EN_ID    : STD_LOGIC_VECTOR (7 downto 0) := X"83";
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- Decoder for updating output registers
-- Register updates depend on rising clock edge and asserted load signal
-------------------------------------------------------------------------------
outputs: process(CLK, s_load, s_port_id)
begin
   if (rising_edge(CLK)) then
      if (s_load = '1') then

         if (s_port_id = LEDS_ID) then
            r_LEDS <= s_output_port;
         elsif (s_port_id = SEGMENTS_ID) then
            r_SEGMENTS <= s_output_port;
         elsif (s_port_id = DISP_EN_ID) then
            r_DISP_EN <= s_output_port(3 downto 0);
         end if;

      end if;
   end if;
end process outputs;
-------------------------------------------------------------------------------
```

Figure 19-7: VHDL model for the "wrapper output hardware" module in Figure 19-7.

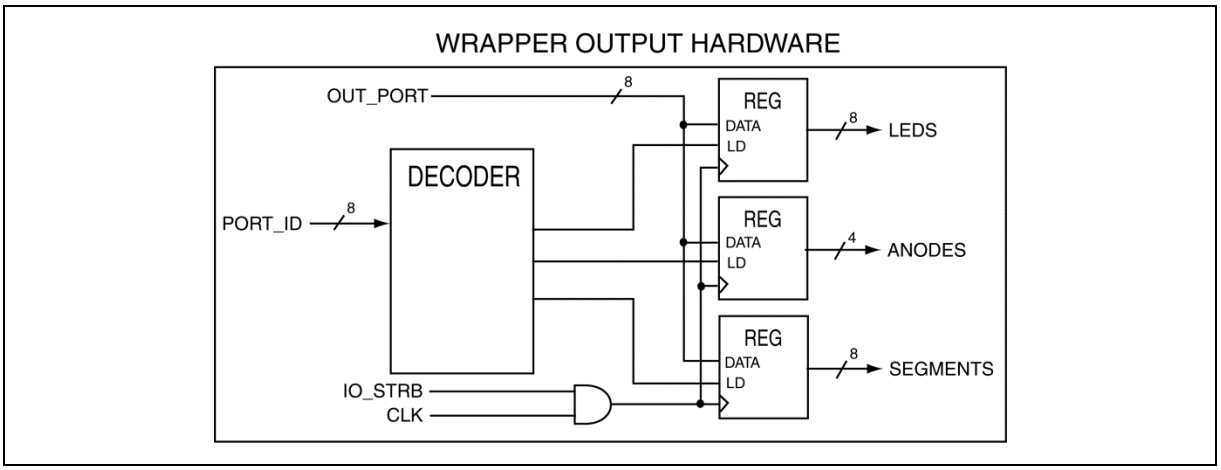


Figure 19-8: Possible hardware induced by Wrapper Output Hardware module in Figure 19-5.

The notion of the wrapper is important for two reasons. First, because it helps you understand how to interface a softcore MCU such as the RAT MCU with a development board. Second, because it helps you understand how the IN and OUT instructions work with the RAT MCU and the RAT wrapper to actually implement I/O on the RAT MCU. For these reasons, you should really make sure you understand everything all aspects of the RAT MCU wrapper.

19.4 RISC vs. CISC

Out there in computer land, you'll find that people attempt to model computer architectures as one of two different types. Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). The names probably meant something at one time, but there now something that you should not take literally.

Since the dawn of computers, or even before, there has been an ongoing argument of which architecture is "better": RISC or CISC? To understand the parameters of the RISC vs. CISC argument, you must understand the current accepted differences between RISC and CISC architectures. Table 19.1lists the commonly accepted characteristics and differences between RISC and CISC architectures.

RISC Architectures	CISC Architectures
• All instructions execute in the same number of clock cycles • Instructions are relatively simple compared to CISC architecture • System clock speed is relatively fast • Instruction set has relatively few addressing modes • Has a relatively large register file	• Instructions require varying numbers of clock cycles for execution • Instructions are relatively complex compared to RISC architectures • System clock speed is not overly fast • Instruction set has relatively many addressing modes • Has a relatively small register file

Table 19.1: The main accepted differences between RISC and CISC architectures.

Generally speaking, the instructions on a RISC machine relatively simple, which allows them to execute in a few number of clock cycles. Conversely, instructions on a CISC machine can be complicated and thus require more instruction cycles or longer clock periods to execute. In the end, to complete the same task, a program written for a RISC architecture are longer because the instructions are simple so there must be more of them to complete the same task. The same program functionality implemented on a CISC architecture will be shorter because each

instruction can generally do more stuff. Nevertheless, because each instruction is doing more stuff, the system clock period must be longer. The general thought here is that it takes more instructions for a RISC computer to perform a given task than it does for a CISC computer. However, the RISC instructions, because of their simplicity, allows for a higher clock speed. This is a classic trade-off in computer-land.

19.5 Levels of Memory

A typical computer system has many type of memory and many memory entities. Even a simple computer such as the RAT MCU contains three relatively large structured memories (prog_rom, scratch ram, and register file), several special register memories (stack pointer, program counter), and a few flip-flops (condition flags, interrupt flag). Computer architects are always attempting to classify items in computer architecture, and one of the primary targets is the structured memories. For this discussion, we're interested in the two writable structured memories in the RAT MCU architecture, which are the register file and the scratch RAM.

Computer architect often speak of "levels of memory" in the context of structured memories in a given computer system. In this way, they consider a typical computer model to have multiple levels of memory. We differentiate these different levels of memory by one thing: their access times (how fast you can read and write to them). According to this definition, we consider the RAT MCU as having two levels of memory: the register file and the scratch RAM.

We consider the register file to be a lower level of memory than the scratch RAM because we can access the data in the memory faster than we can access the memory in the scratch RAM. Recall that all useful operations in the RAT MCU occur with instructions that access the register file. This means if we have data in a register, we're immediately ready to operate on it. This differs from data in the scratch RAM in that if we want to operate on data stored in the scratch RAM, we first must load it into a register in the register file. The notion of loading the value into the register file requires an extra instruction, and thus accessing the scratch RAM is "slower" than accessing the register file. Moreover, since accessing data in the scratch RAM is slower than accessing data in the register file, we consider the scratch RAM a higher-level of memory than the register file.

A typical computer system such as you laptop computer has many different levels of memory. For example, the lowest-level of memory may be some type of general purpose register as with the RAT MCU. More complex computer architectures will have many more levels of memory; so for something like your laptop, the hierarchy of memory starts with registers, then goes to cache, scratch memory, RAM, hard-drive, etc.

The general thought with levels of memory is that the lower the level, the more expensive it is, the faster it is in terms of access, and the less your system has of it. You certainly see this with the RAT MCU; there are several reasons why the register file memory is more "expensive" than the scratch RAM. On the other end, memory in hard drive is cheap, plentiful, and requires a relative long time to access. Each level of memory has its place in a computer system; exactly what place that is, is something computer architects deal with constantly.

19.6 Chapter Summary

- Despite the RAT MCU being a relatively simple device, there are timing issues associated with the instructions that you must understand in order to understand the overall operation of the RAT MCU. If you're just a programmer, you don't really need to understand these timing details. But if you know/understand anything regarding the underlying RAT MCU hardware, you must understand basic timing issues.

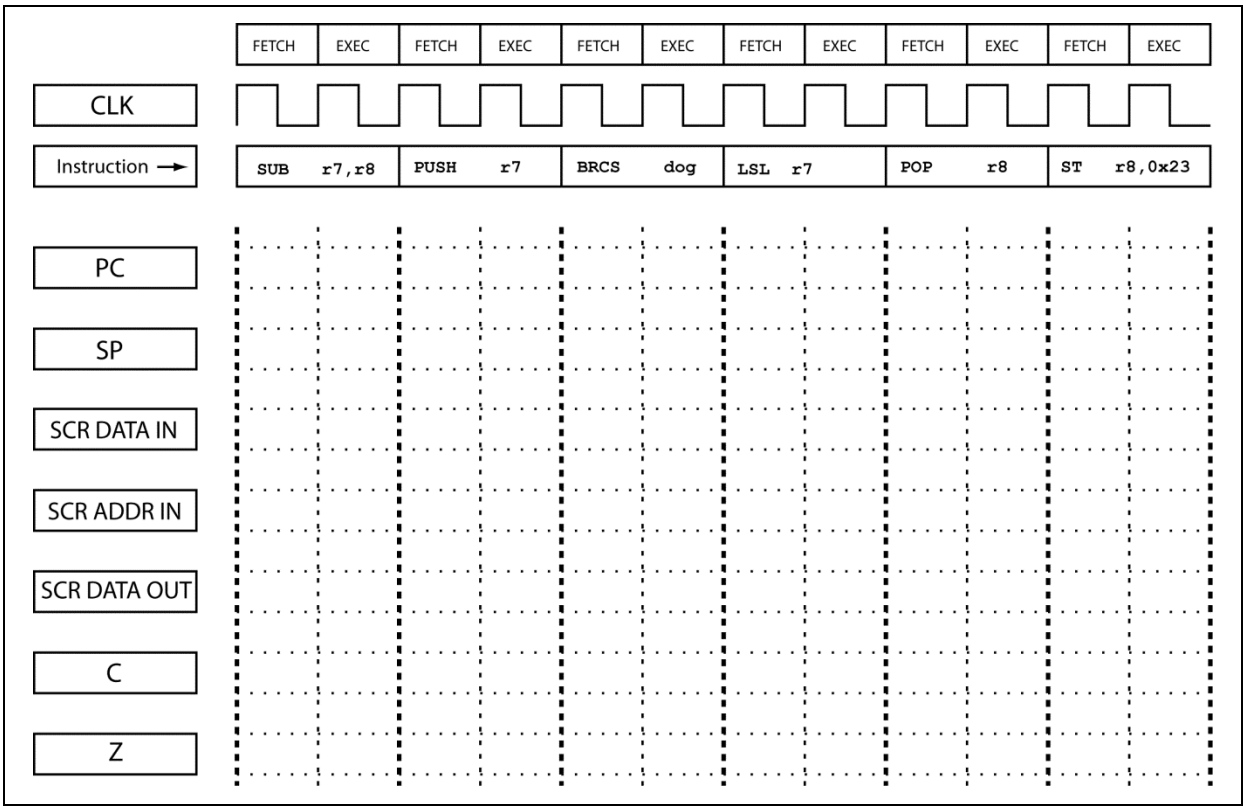
- The RAT MCU "wrapper" provides an interface between the RAT MCU and a development board. The current RAT MCU wrapper is a relatively simple VHDL model that interfaces the RAT MCU I/O with the various input (such as buttons and switches) and output devices (such as LEDs) on the development board. The highlights of the wrapper include a MUX for the development board's inputs and a decoder and register for the development board's outputs.

- Reduced instruction set computers (RISC) and complex instruction set computers (CISC) are the two main classifications that we try to place computer architectures into. RISC architectures have instructions that execute in the same number of clock cycle, relatively large register files, few addressing modes, and relatively simple instructions. CISC architectures have all the opposite characteristics. Programs written for RISC architecture generally have more instruction than the same program for a CISC architecture, but the RISC instructions generally execute in a smaller amount of time.

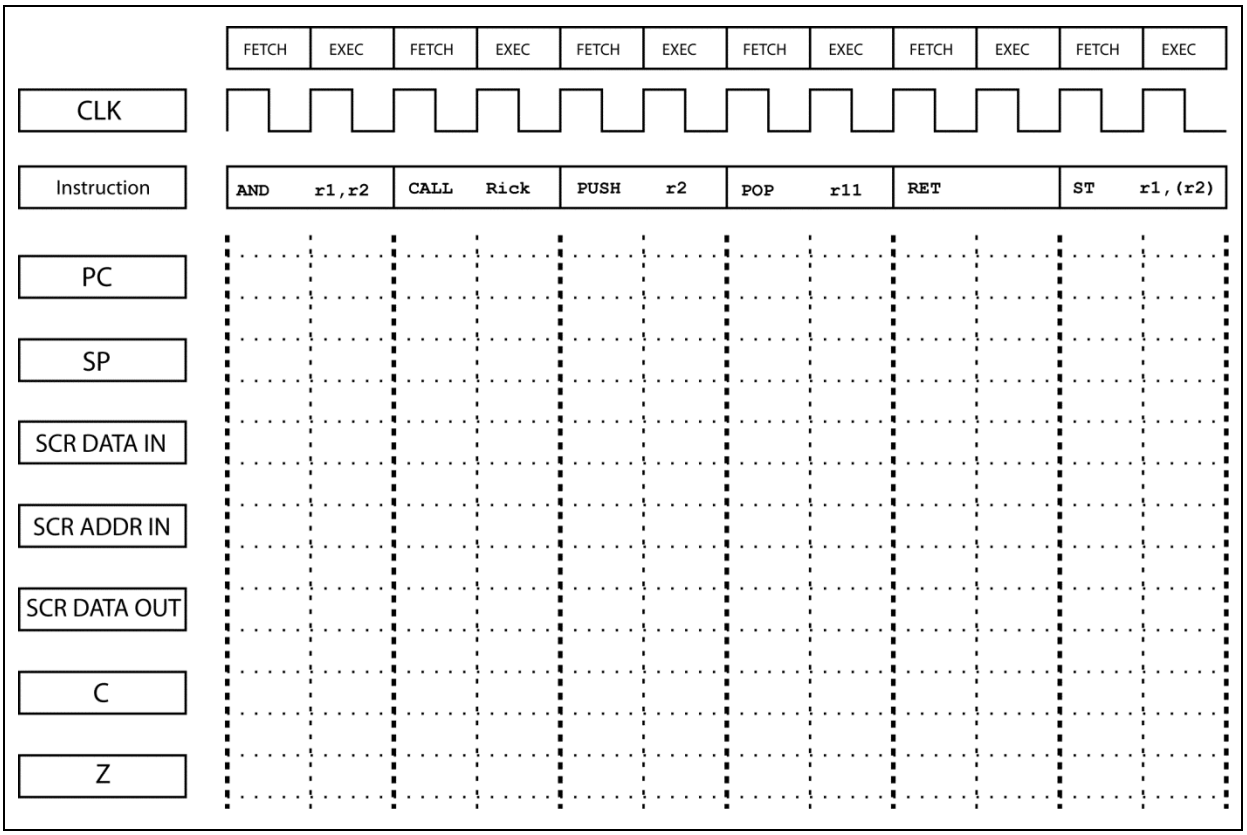
- The notion of levels of memory refers to how fast a system can access (read and write) that memory. Generally speaking, lower levels of memory have faster access times, are but lower storage capacity than higher levels of memory.

19.7 Chapter Exercises

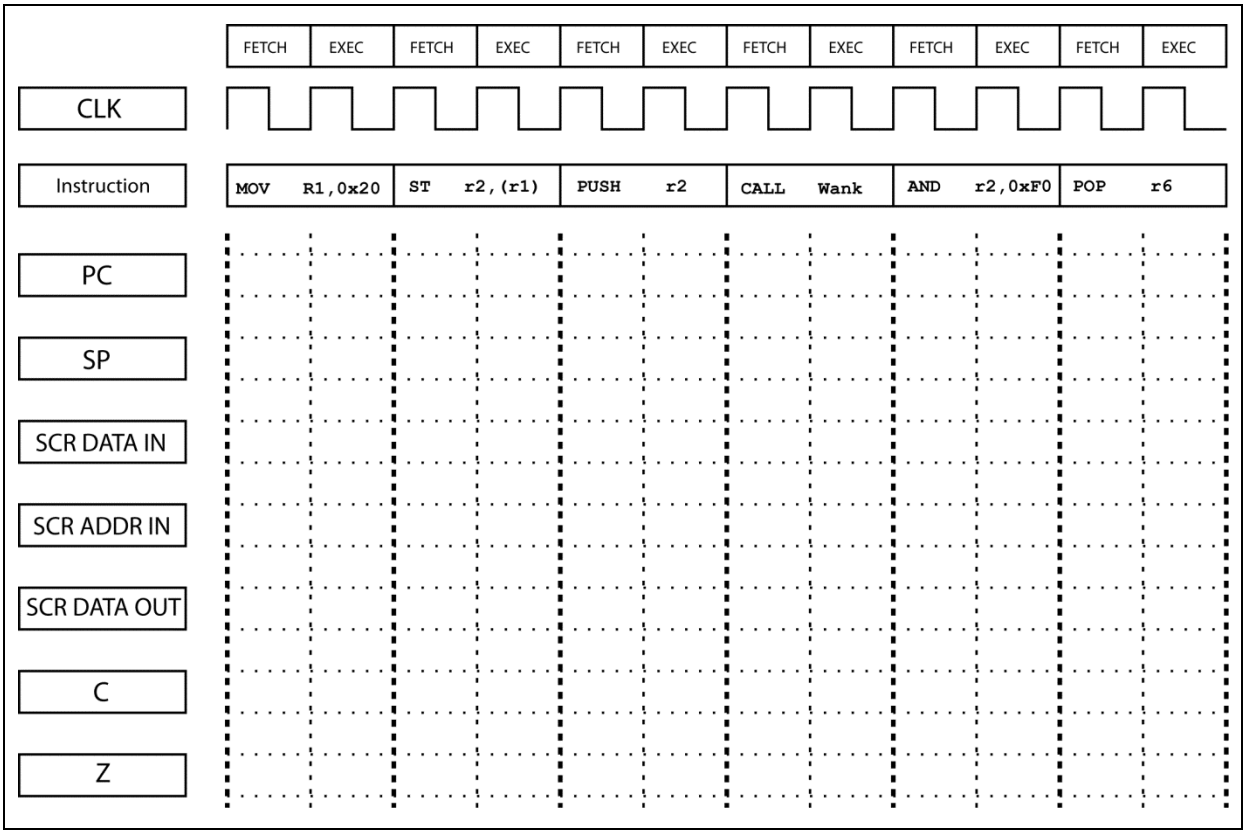
1. Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x100, SP = 0xD0, C = ’0’, and Z = ‘0’. Additionally,0x200 is the address value associated with the “dog” label, r7 holds a value of 0x40, and r8 holds a value of 0x41.



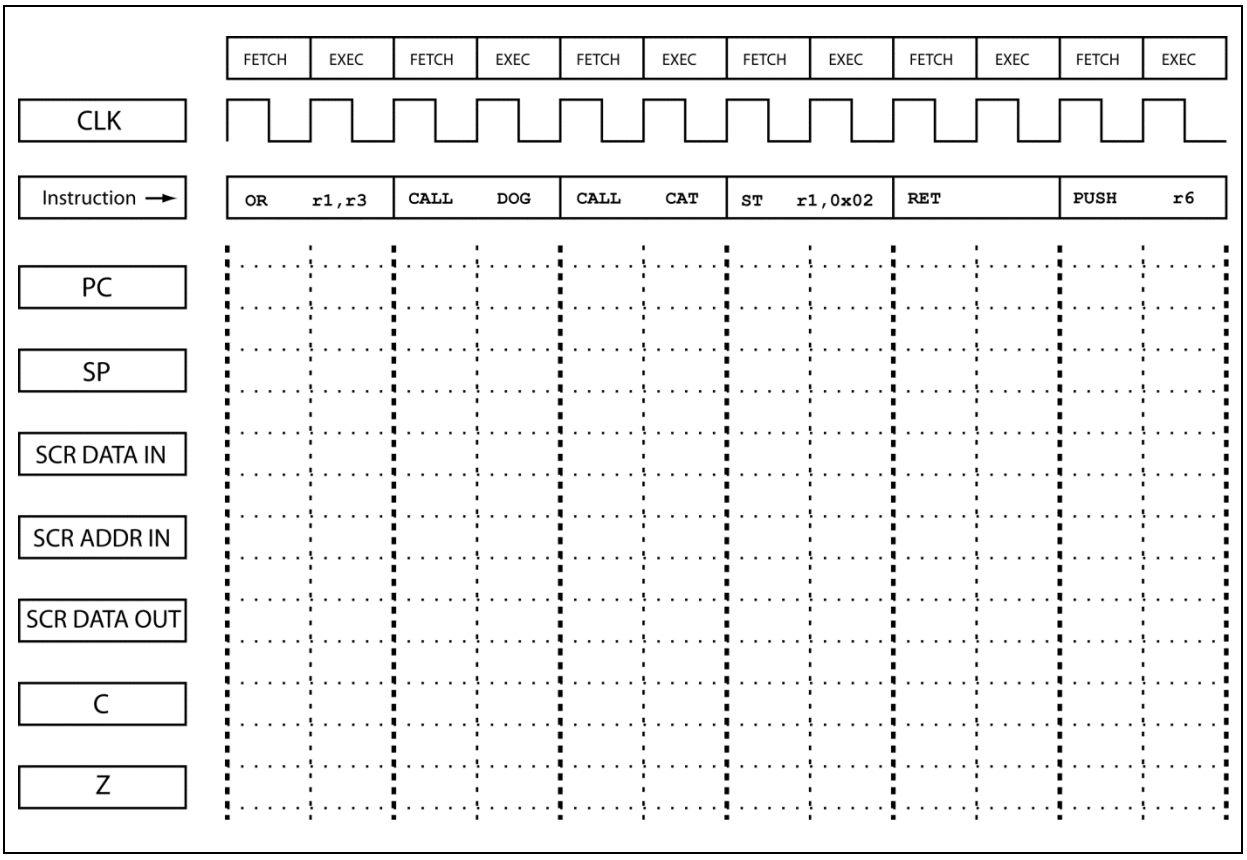
2. Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x0B9, SP = 0xF6, C = '1', and Z = '0'. The "Rick" label has a value of 0x0CA, r1 holds a value of 0x55, r2 holds a value of 0xAA, and r11 holds a value of 0xDF.



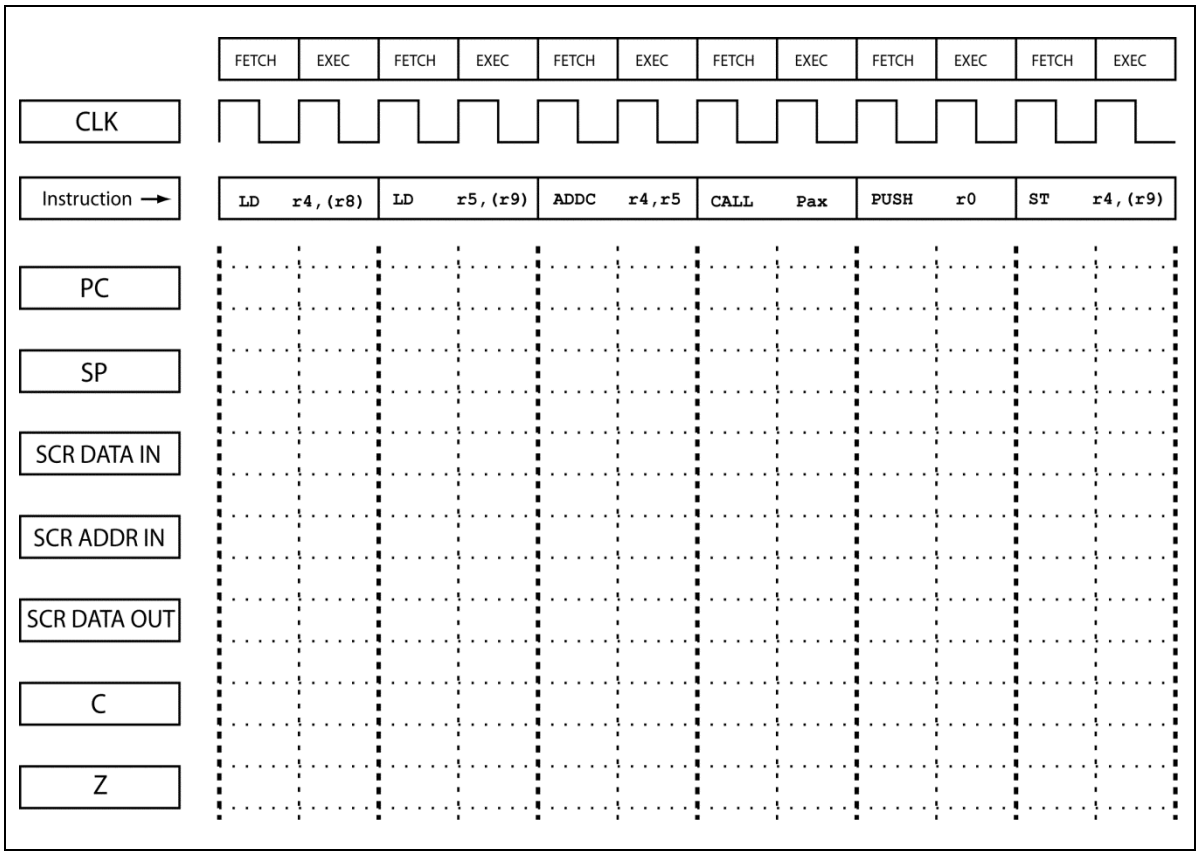
3. Complete all aspects of the following timing diagram for the given six instructions. For this problem assume initial values of PC = 0x0AA, SP = 0xEA, C = ’1’, and Z = ‘1’. Additionally, the “Wank” label has a value of 0x0DA, r1 holds a value of 0x47, r2 holds a value of 0xCC, and r6 holds a value of 0x33.



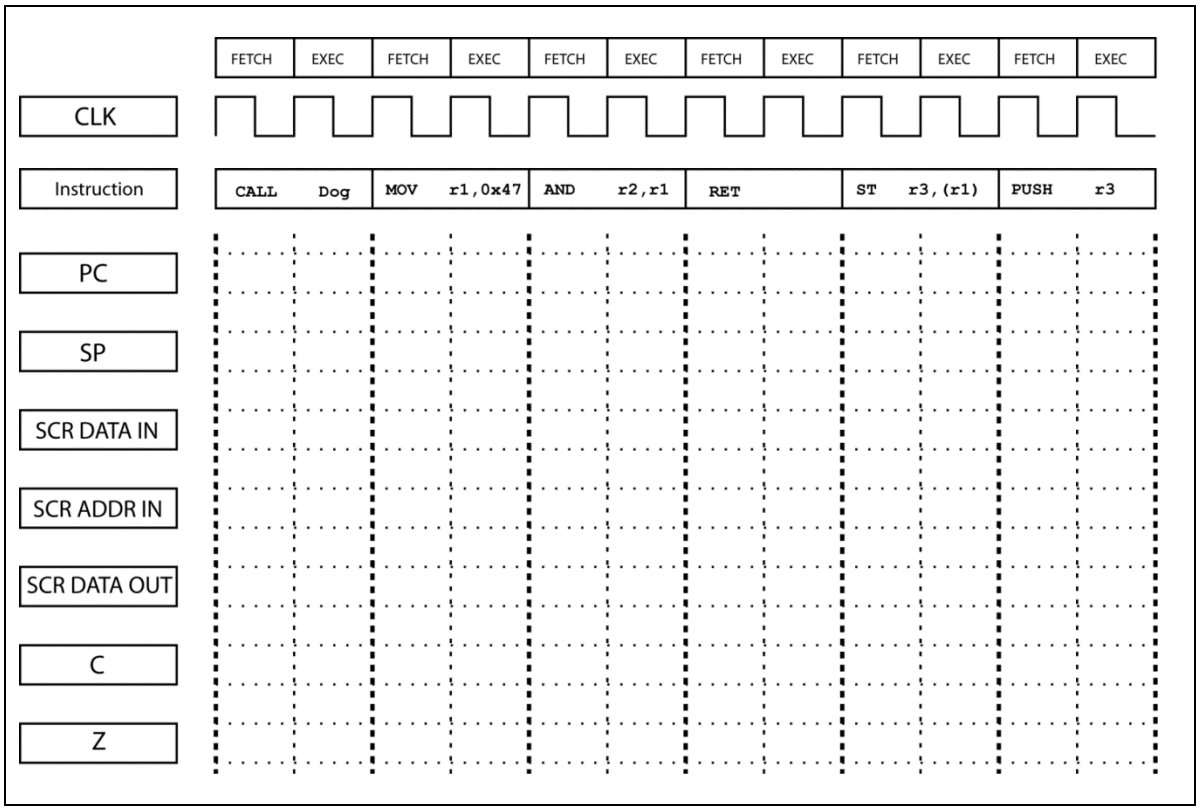
4. Complete all aspects of the following timing diagram for the given six instructions. For this problem assume the following initial values: r1 = 0x46, r3 = 0x01, r6 = 0x87, PC = 0x040, SP = 0xCC, C = ’1’, Z = ‘1’, the “DOG” label has a value of 0x0A5, the “CAT” label has a value of 0xE4, the value in scratch RAM addresses 0x02 is 0xEE.



5. Complete all aspects of the following timing diagram for the given six instructions. For this problem assume the following initial values: r0 = 0xF2, r8 = 0xAF, r9 = 0xB0, PC = 0x090, SP = 0xB0, C = '0', Z = '0', the "Pax" label has a value of 0x033, the values in scratch RAM starting at addresses 0xAF is 0x89 followed by 0x77.



6. (15) Complete the timing diagram below for the given instructions using the provided initial values. You only need to show SCR DATA IN, SCR ADDR IN, and SCR DATA OUT when the instruction accesses scratch RAM. r1 holds a value of 0xFF, r2 holds a value of 0xB8, r3 holds a value of 0xAA, PC = 0x050, SP = 0xCC, C = '1', Z = '0', "DOG" label has a value of 0x132, Scratch RAM addresses 0x02 = 0xAA, Scratch RAM addresses 0x47 = 0xC7.



7. Briefly but completely describe the purpose of the RAT MCU wrapper.

8. Would it be possible to use the current RAT Wrapper for another softcore MCU. Briefly but completely explain your reasoning.

9. Briefly but completely explain the main differences between the input and output portions of the RAT MCU wrapper.

10. The RAT MCU can only control a fixed number of input and output bits.

    - Describe what determines the number of input and output bits can be controlled by the RAT MCU.
    - How many total input bits can the RAT MCU control? Show the calculation for this question.
    - How many total output bits can the RAT MCU control? Show the calculation for this question.

11. Show a completed timing diagram based on the current RAT MCU wrapper that indicates exactly when the data is latched into the output registers.

12. Show the modifications necessary to the wrapper code so that you can add two more input devices: a set of six buttons (map them to address 0x44) and a set of seven switches (map them to address 0x77).

13. Show the modifications necessary to the wrapper code so that you can add three more input devices: a set of five LEDs, another four-digit 7-segment display, and a set of 16 LEDs. You can decide any port addresses you want for this question.

14. In your own words, explain which is better (or if one is not better) RISC or CISC architectures.

15. Modern computer architectures often blur the accepted definitions of RISC and CISC architectures. Briefly but completely explain why that is.

16. If lower levels of memory are faster, briefly but completely explain why there is a need for higher levels of memory.

17. What is a lower level of memory: a tape drive or a hard drive? Briefly explain.

20 RAT Architectural Modifications

20.1 Introduction

As you extend you knowledge regarding computer architecture and assembly language programming, you'll no doubt start questioning some of the design decisions that went into design the RAT MCU. In truth, a countless number of design decisions went into designing the RAT MCU in areas such as the RAT hardware architecture, the RAT Interrupt Architecture, and the RAT Instruction Set Architecture.

As with any digital design, if you stare at it long enough, you'll surely figure out a better approach to the design. This become more true as you gain more digital design experience in that you'll have accumulated more tricks in your digital bag of tricks. The same can be said for your assembly language programming skills.

This chapter allows you to apply your knowledge and skills by asking you to describe various changes to the RAT MCU. The idea here is that you won't be able to make viable modifications unless you understand all aspects of the RAT MCU, including aspects of the RAT assembler as well. If you're not quite at that point, the examples in this chapter will quickly move you along in the direction of complete understanding of the RAT MCU.

Main Chapter Topics

- **RAT HARDWARE ARCHITECTURE MODIFICATIONS:** This chapter outlines hardware modifications in order to achieve various stated design goals.
- **RAT ASSEMBLER MODIFICATIONS:** This chapter outlines changes to the RAT assembler in the context of desired hardware architecture modifications.
- **RAT INSTRUCTION SET ARCHITECTURE MODIFICATIONS:** This chapter outlines changes in the instruction sets in response to proposed RAT hardware architecture changes.

Why This Chapter is Important

This chapter is important because it advances your knowledge of the RAT MCU by outlining possible hardware architecture changes in response to stated design goals.

20.2 RAT Architectural Modifications and Extensions

The section comprises of suggested modifications and changes to the RAT hardware architecture. These changes may also causes changes in the RAT assembler and/or the RAT Instruction Set architecture. These problems represent my best take on these proposed changes. I have no doubt that I have mistakenly omitted important information in these examples. If you discover something that I did not see, than you definitely know the RAT MCU architecture, which is the underlying goal of this text and associated course.

One of the big issues associated with modifations to the RAT MCU hardware involves the notion of the width of the instruction word. Recall that there are five different instruction format and that each instruction

comprises of 18 bits. The RAT MCU's instruction memory represents a major portion of the total memory of the RAT MCU, which means increasing the instruction word length is relatively costly while reducing the instruction length is relatively beneficial in terms of overall RAT MCU memory requirements. Why this is important is because if you "add" something to the RAT MCU, there is a chance the change will require an increase the length of the instruction word. Conversely, if you "remove" something from the RAT MCU, there is a possibility that the change will allow you to reduce the size of the instruction words. In this context, "adding" represents items such as including new instructions, increasing the size of some RAT MCU hardware feature, etsc. And no surprise that "removing" something includes getting rid of instructions, reducing the size of something on the RAT MCU etc.

Recall that only the reg-immed instruction format contains unused bits in the instruction word. This fact, generally speaking, makes the reg-immed instruction format the limiting case for "adding" or "removing" something from the RAT MCU. This means that if I add something, I'll first check to see if the addition causes issues with the reg-immed format. The reg-immed format includes five op-code bits, five bits for the register address, and eight bits for the immediate value. If for example, you reduce the size of the register file by one half, you'll only need four bits to encode the register address, and thus can reduce all instruction formats by one bit. The results will be a memory savings of 1024x1 because the instrution format now only needs to be 17 bits to support the entire instruction set. And as you can probably imagine, if I increase the register file to 64 registers, you can use the same arguments to state that the reg-immed format would require six bits to encode the register address and thus the instruction width of all instructions would grow to 19 bits.

The above paragraph states general guidelines or a starting point for working with changes to the RAT MCU. In reality, you must consider each addition of removal individually in order to ascertain its full ramifications. For example, the reg-immed instruction contains both a register value and a immediate value, but it does not contain a program counter address as to the call-type and branch-type instructions. For example, any changes to the size of the prog_rom will necessarily change the width of the address field in these instructions. Under those conditions, the call-type instructions would be the instruction format you would examine in order to ascertain whether the hardware change in quesiton would cause a change in the size of the instruction word. In this case, the reg-immed type instrution is therefore not the limiting case. So break out the assembler manual when you paw through these examples.

Example 20.1: Adding a NBSWP Instruction

Add an instruction that swaps the nibbles in a register: NBSWP. For this problem, do the following:

Example: **`NBSWP  r23`**

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) We need to modify the ALU to support the new instruction. There is currently space for one extra instruction using the current four ALU select signals. The control unit needs to modification so that it recognizes this new instruction and sends out the appropriate control signals.

b) The assembler needs to be modified to recognize this new instruction. This is a reg-type instruction and there is plenty of code space for it, so the instruction word width does not increase.

c) Although we added a new instruction to the control unit, we did not add a new state. Since the memory in the control unit is based on the number of states, we did not change the amount of memory in the control unit. The ALU is a combinatorial module, so the ALU-based changes did not increase memory either.

d) This could be a useful instruction when dealing with packed BCD numbers (an 8-bit value that contains two BCD numbers).

Example 20.2: Adding a WRPC Instruction

You need to create a new instruction: WRPC. This instruction allows me to write a value from the scratch RAM directly into the program counter. For this problem, do the following:

Example: **`WRPC immed_val`**

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: The first thing to note about this problem is that we are "adding" something to the RAT MCU. This means that there is a possibility that the width of the instruction word will grow.

a) Some of the hardware is already in place for this change: there is a path from the output of the scratch RAM to the PC MUX to support RET-type instructions. The limitation of this new is that the only possible method to place 10-bit data in the scratch RAM is from PC values. The control unit needs to change to recognize this new instruction and send out the appropriate control values.

b) We need to change the assembler so that it recognizes this new instruction. This requires that we add a new immed-type instruction with its associated operational and field codes.

c) Since the new instruction is an immed-type instruction, there is plenty of code space available so we don't need to extend the instruction width. We modified the control unit, but we did not add any new states. Since the number of states in an FSM determines the amount of FSM memory, we effectively do not alter the control unit's memory requirements. In the end, there are no changes in memory requirements.

d) This instruction would be useful to allow the changes to the PC under direct program control. The current RAT MCU instruction set only allows PC mods (other than increments) under indirect control using the various program flow-type instruction (CALL, RET, and branches). There is of course an issue of being able to transfer 10-bit values from the register file to memory, but that is a topic for another problem.

Comment: Another possible form of this instruction could use register indirection to address the scratch RAM. Once again, the hardware and routing is currently in place to immediately support this change and the reg-type instruction also has many unused bits. The form of this instruction would be:

```
WRSP    (r13)
```

In this case, the current reg-type instruction contains unused bits so the only change required to the assembler would be to support this alternative form of the new WRSP instruction. The other parts of the solution are similar.

Example 20.3: Extending the Scratch RAM Size

The RAT MCU scratch RAM resources are not adequate for your needs; you must increase the size of the scratch RAM to 512 locations rather than the current 256. For this problem, do not make changes to the register file or ALU. For this problem, do the following:

e. describe the changes you need to make to the RAT hardware to support this change

f. describe any changes you need to make to the RAT assembler to support this change

g. state the change in RAT memory requirements to support this change

h. describe why this change could be useful

Solution: The first thing to note about this problem is that we are "adding" something to the RAT MCU. This means that there is a possibility that the width of the instruction word will grow. We'll be looking for that as we solve this problem.

a) First, note that the width of the scratch RAM s not changing: the total scratch RAM size is going from 256 x 10 to 512 x 10. The first change required is to increase the width of the stack pointer from eight to nine bits. This new 9-bit address has two ramifications that you'll have to work around. First, the WSP instruction and the indirect forms of the LD & ST instructions no longer work properly, meaning they would not be able to access 512 different values because the width of the register file's registers is eight bits. Depending how the register file's output is connected to the address inputs of the scratch RAM, you could either address the 256 lower addresses in scratch RAM if you connected the register file output to the lower eight bits of scratch RAM address of every other address in scratch RAM is you connect the register file's output to the upper eight bits of scratch RAM address (and set the LSB to '0'). For this problem, you do have an option of what to do about the immediate forms of the LD & ST instructions. You could extend the address field of the LD & ST instruction (immediate form only) to nine bits to allow it to access each address in scratch RAM. Second, the reg-immediate type LD & ST instructions would need an extra bit for the 9-bit address. Since reg-immediate instructions have no unused bits in the instruction format, the nominal width of the instruction needs to increase from 18 to 19 bits.

b) This is an interesting problem because we have some options. If we decided to allow the LD & ST instructions to access all 512 locations in scratch RAM, we have to increase the width of the instruction word from 18 to 19 bits. In this case, we have to significantly modify the assembler, as every instruction needs a new format to support the 19-bit instruction word length. If we opted to not allow LD & ST immediate instructions to access as 512 locations in scratch RAM, then the assembler would not require any changes.

c) For all cases, we need to increase the size of the stack pointer from eight to nine bits. The new size of the scratch RAM is 512x10 for an increase of 2560 bit. If we opted to increase the width of the instruction word, we have a change in 1024 bits.

d) This would be a useful instruction because we could save more data in scratch RAM using PUSHes. It also allows us to nest subroutines from 256 levels to 512 levels. LD & ST immediate instructions would not have full access to scratch RAM data unless we increased the width of the instruction word.

Device	Size	Number of Bits	New
prog_rom	1024 x 18	18432	(?)1024x19=19456
Register file	32 x 8	256	256
Program counter	10	10	10
Stack pointer	8	8	9
Flags (I,Z,C,shZ,shC)	5 x 1	5	5
Scratch RAM	256 x 10	2560	512x10=5120
	TOTAL	**21,271**	**24,856**

Example 20.4: Reducing the Scratch RAM Size

You need to reduce the size of the scratch RAM from 256x10 to 128x10. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change

b. describe any changes you need to make to the RAT assembler to support this change

c. state the change in RAT memory requirements to support this change

d. describe why this change could be useful

Solution: This problem asks you to remove something from the RAT MCU, which means there is a possibility that the width of the instruction word will decrease.

a) The size of the scratch RAM decreases from 256x10 to 128x10. This means that the width of the stack pointer changes from eight to seven bits. We need to consider the notion of whether this change results in a reduction of the instruction word length. This change would lie in reducing the number of bits in the reg-immed-type instruction. The immediate field in reg-immed-type instructions serves two purposes. Although it does serve as an address for LD & ST instructions (reg-immed formats), the field also serves as a way to place 8-bit data into the register file for ALU-based reg-immed-type instruction. Because we are not changing the width of the register file data, we can't alter the width of the immediate field in the reg-immed-type instructions. The hardware changes to recognize only seven of the eight current bits in the immediate value for the address scratch RAM. The hardware needs to change to use only seven of the eight bits for the reg-reg-type of the LD & ST instructions.

b) The assembler needs no changes because we are not adding any instructions and we are not increasing the width of instructions words. You could be clever and add an error message for LD & ST instructions that had immediate operands using values greater than 127.

c) The two changes in memory requirements are the scratch RAM changing from 256x10 to 128x10, and the stack pointer changing from eight bits to seven bits.

d) This change maked the RAT MCU footprint smaller, which could potentially make the RAT MCU faster and operate with lower power requirements.

Device	Size	Number of Bits	New
prog_rom	1024 x 18	18432	18432
Register file	32 x 8	256	256
Program counter	10	10	10

Stack pointer	8	8	7
Flags (I,Z,C,shZ,shC)	5 x 1	5	5
Scratch RAM	256 x 10	2560	128x10=1280
	TOTAL	**21,271**	**19,990**

Example 20.5: Adding LSL0 and LSR0 Instructions

Add two new instructions to the RAT MCU instruction set: LSL0 and LSR0. These instructions are logical shift left and right instructions, respectively. They differ from the current LSL and LSR instructions because these instructions place '0' into the new bit position resulting from the shifts rather than the value of the C flag being placed at those locations. The C and Z flags for these instructions are the same as for the LSL and LSR instructions. For this problem, do the following:

Example: `LSR0 r11`

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) The ALU needs to be modified to support these new instructions. There is currently space for one extra instruction using the current four ALU select signals, but we can combine either the CMP/SUB or the AND/TEST instructions to create space for the two operations. The control unit needs to modified so that it recognizes these two new instructions and sends out the appropriate control signals.

b) The assembler needs to be modified to recognize these new instructions. These are both reg-type instructions and there is plenty of code space for more reg-type instructions, so the instruction word width does not increase.

c) Although we added a new instruction to the control unit, we did not add a new state. Since the memory in the control unit is based on the number of states, we did not change the amount of memory in the control unit. The ALU is a combinatorial module, so the ALU-based changes did not increase memory either.

d) The reason why this instruction is potentially important is that the current implementations of the LSL and LSR instructions utilize the C flag in such a way as to cause issues with the multiplication and divide (left-shift and right-shift, respectively). The current usage requires that you issue a CLC before every shift instruction, or, clear the C-based shifted bits shifted in once you finishing your desired amount of shifting. The addition of this instruction to the RAT MCU instruction set would therefore save writing code to ensure the bit shifted in was in fact zero.

Example 20.6: Adding Four Special Shift-Type Instructions

Add four new instructions to the RAT MCU: LSR0, LSL0, LSR1, LSL1. These are logical shift left/right instructions that insert 1's or 0's into the operand register instead of shifting in the C flag. The numeric value in the instruction mnemonic is what shifts into the register's MSB (right shifts) or LSB (left shifts). For these instructions, the C and Z flags operate as the do with the LSL and LSR instructions. For this problem, do the following:

Example: **LSL0 r8**

i. describe the changes you need to make to the RAT hardware to support this change

j. describe any changes you need to make to the RAT assembler to support this change

k. state the change in RAT memory requirements to support this change

l. describe why this change could be useful

Solution: The first thing to note about this problem is that we are "adding" something to the RAT MCU. This means that there is a possibility that the width of the instruction word will grow. We'll be looking for that as we solve this problem.

b) The main issue with this problem is that we're adding four new ALU-based instructions. The ALU currently performs 15 operations, two of those operations are repeated (CMP/SUB & AND/TEST). We could combine those instructions to bring the operation count down to 13, but we're still adding four instructions, which brings the total count to 17 instructions. Since there are only four ALU select bits, we need to increase this to five bits so it can handle 17 instructions. The first change we need to make then is to add hardware to the ALU to support these four instructions. Next, we need to make two changes to the control unit. First, we need to extend the width of the ALU select signal from four to five bits. Second, we need to modify the control unit to that it recognizes the opcodes associated with these four instructions and then sends out the correct control bits.

c) We need to change the assembler so that it recognizes these four new instructions. This requires we add four new reg-type instructions with their associated operational and field codes.

d) Since the four new instructions are reg-type instructions, there is plenty of code space available so we don't need to extend the instruction width. We added stuff to the ALU and the control unit, but the ALU is combinatorial so we did not add memory. We modified the control unit, but we did not add any new states. Since the number of states in an FSM determines the amount of FSM memory, we effectively do not alter the control unit's memory requirements.

e) These instructions would be useful in cases where we need to set of clear the carry flag prior to executing a shift left or right instruction. Thus, we could effectively save an instruction in many cases, which is significant in cases where the instruction is in an iterative loop.

Comment: The above solution was how I saw the problem. I actually like the solution of a student (John T.) better. In brief, John mentioned that we could add a MUX in front of the C input to the ALU. This would effectively be a 4:1 MUX, which allows instructions to input either the current C value, a '0', or a '1' to the MUX. This allows instructions to select which value goes into the new register value. This solution requires the addition of a 4:1 MUX, but does not require changes to the ALU. The control unit requires changes to support these instructions, but also two new signals as select inputs for this MUX. I sure wish I had thought of that; borrowing ideas from smart students is part of being an instructor.

Example 20.7: Modifying the Context Saving Mechanism

Modify the RAT MCU such that you can save/restore the C & Z flags on the stack using two new instructions: PUSHCZ & POPCZ. For this problem, the interrupt architecture no longer saves context in the interrupt cycle or restores context with RETID & RETIE. For this problem, do the following:

Example**:** **PUSHCZ**

Example**:** **POPCZ**

m. describe the changes you need to make to the RAT hardware to support this change

n. describe any changes you need to make to the RAT assembler to support this change

o. state the change in RAT memory requirements to support this change

p. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) The first thing to note is that we'll be saving the C & Z flags to the stack, which means we can remove the shadow C & Z flags. Next thing to note is we now need a datapath from the C & Z flags to the scratch RAM inputs, and from the scratch RAM outputs to the C & Z flag inputs. The scratch RAM requires both address and data value for writes (PUSHCZ), and only an address for reads (POPCZ). For PUSHCZ, we need to add a datapath from the output of the flag registers to the data input of the scratch RAM. This then requires that we change the scratch RAM's address MUX from a 2:1 to a 4:1 MUX, which requires that we add another control signal to the MUX. The POPCZ instruction needs a datapath from the output of the scratch RAM to the inputs of the C & Z flags. There is currently a MUX in front of both the C & Z flags left there from the shadow flags; we retain these MUXes to allow the loading of the C & Z flags to input either the output of the ALU of the output of the scratch RAM for the POPCZ instruction. The control unit needs to change to recognize these two new instructions. We also need to add another control signal to the control units output to handle the new select signal for the 4:1 scratch RAM data MUX.

b) We need to change the assembler so that it recognizes these new instructions. This requires we add a new none-type instruction with their associated operational and field codes to the assembler.

c) Since the new instructions are non-type instructions, there is plenty of code space available so we don't need to extend the instruction width. We modified the control unit to support these instructions, but we did not add any new states. Since the number of states in an FSM determines the amount of FSM memory, we effectively do not alter the control unit's memory requirements. In the end, there are no changes in memory requirements.

d) The instructions would be useful because it allows us to nest interrupts. The current RAT MCU architecture does not support the nested interrupts because the context-saving mechanism overwrites the shadow flags after the first level of interrupt nesting. The possible drawback of these instructions is that the programmer is 100% responsible for context saving (via PUSHCZ and PUSHes of registers) and context restoration (POPCZ and POPs). Note that the first instruction in an ISR should not be PUSHCZ and the last instruction before the return from interrupt instruction (RETIE or RETID) should be POPCZ.

Example 20.8: Adding a Second Interrupt to the RAT MCU

Describe the required changes to add one more interrupt to the RAT MCU. Assume the operation of the new interrupt (interrupt architecture) would be the same as the current interrupt. State any other assumptions you need for your particular solution. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: There are many valid solutions to this problem. Thus, any solution you provide will need to also include the any assumptions you make in your solution. The solution below is the full-featured solution; I'll add a comment at the end regarding other valid solutions. Note that this problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) We need to first and another interrupt flag; this flip-flop would have all the same features as the current I-flag. We also need to include a second AND gate to support the second interrupt. This change would also require that we add a second interrupt input to the control unit. Additionally, since we're adding another interrupt, we need to add a second interrupt cycle, which entails adding another state to the control unit's underlying state machine. The program counter's MUX would need to use the unconnected data input to hardcode a second interrupt vector, such as 0x3FE. For this problem, I'm opting to add full program support for this interrupt (as opposed the MCU sharing the current interrupt based instructions). This path includes two versions of both RETIE, RETID, SEI, and CLI instructions, meaning I need to also change the control unit to recognize and send out the correct control signal for these new instructions.

b) Because we're adding four new instructions, we need to change the assembler to recognize these instructions. All of the added instructions are non-type instructions.

c) We added a new I-flag, which requires one-bit of memory. We added four new instructions to the control unit, which does not affect the number of states in the control unit's FSM so does not cause a change in memory requirements. The four new instructions are none-type instructions; there is plenty of space for new instructions of this type so we don't need to change the width of the instruction word. We did add another state to the control unit to support the second interrupt cycle, but this does not necessarily change the memory requirements as we increased the number of FSM state from three to four. The FSM requires the same number of bits (in the minimum case) to store both three and four states.

d) The RAT MCU is extremely limited having only one interrupt. This change would extend the usefulness of the RAT MCU by having two interrupts to deal with.

Comment: One other valid solution would be to use the same four interrupt-based instructions to handle the second interrupt. This solution means that you would not have independent control of the two interrupts. In other words, you could only enable or disable both interrupts at the same time. Although this approach would be overall much more simple, it would sacrifice a significant amount of functionality.

Example 20.9: A Leaner and Meaner RAT Architecture

You want to reduce the RAT MCU footprint by reducing the register file to eight registers, reducing the size of scratch RAM to 128 registers, and reducing the prog_rom to only hold 512 instructions. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change

b. describe any changes you need to make to the RAT assembler to support this change

c. state the change in RAT memory requirements to support this change

d. describe why this change could be useful

Solution: This problem asks you to remove something from the RAT MCU, which means there is a possibility that the width of the instruction word will decrease.

a) The new format requires has only eight registers in the register file; this means that the register file addresses will now be three bits rather than eight. The scratch RAM is now 128x10 rather than 256x10, which allows us to reduce the size of the stack pointer. This change could potentially reduce the width of register file data as the addresses to scratch RAM are now only seven bits. However, this change can't happen because the immediate field in instructions still needs to be eight bits in order to support ALU-based reg-immed instructions. The program memory is now 512x??, which has half the locations of the standard RAT MCU architecture, thus we can reduce the program counter from ten to nine bits. We don't know about the width of the instruction word.

b) The limiting case in instruction sizes is the reg-immed-type instruction; the reduction in width of this instruction format will decide the instruction lengths of other instruction formats. Because we now only have three bits in the register file's address field, the overall width of the reg-immed-type instruction will decrease to 16-bits from the standard 18 bits. Because the instruction widths for all instructions are changing, the assembler will require changes to all instruction formats, which represents a significant set of changes.

c) The register file's memory will be 8x8, down from 32x8. The scratch RAM memory will decrease to 128x10 from 256x10. The stack pointer decrease from seven bits from eight bits. The program memory will decrease from 1024x18 to 512x16.

d) These changes will give the RAT MCU a smaller footprint, which means it could potentially run faster and consume less power.

Device	Size	Number of Bits	New
prog_rom	1024 x 18	18432	512x16=8192
Register file	32 x 8	256	8x8=64
Program counter	10	10	9
Stack pointer	8	8	7
Flags (I,Z,C,shZ,shC)	5 x 1	5	5
Scratch RAM	256 x 10	2560	128x9=1152
	TOTAL	**21,271**	**9429**

Example 20.10: A New Context Saving Mechanism for the RAT

For this problem, in addition to the current context saving mechanism, you want the RAT MCU to automatically store every register in the register file into a special shadow register file upon receiving an interrupt (context saving) and copying the registers back to the real register file upon return from the interrupt (restoring context). For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change

b. describe any changes you need to make to the RAT assembler to support this change

c. state the change in RAT memory requirements to support this change

d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

This problem has many possible solutions; we'll describe one approach that heavily involves the control unit, and later mention another approach that reduces the overall control unit involvement but requires extra hardware.

a) First, we need to add the extra register file, which is a 32x8 RAM. The main idea being this change is to have the contents of the main register file copy back and forth to the shadow register file. When the RAT MCU receives an interrupt, the hardware must copy the contents form one RAM to the other. This requires a datapath from the main register file to the shadow register file. Either the control unit or an outside 5-bit counter generated the addresses. The control unit also required some additional control signals, as it is a counter that needs some level of control. If we used a counter external to the control unit, the control unit would control the external counter. For this solution, we'll add 32 or so extra states to the control unit, which represents one state of each register file location. The control unit would them step through all 32 states to copy the register file's contents when it saves context. For context restoration, we first need to MUX the main register file's input data so that it can choose to load register value from either the places it currently loads from or from the shadow register file. There is already a MUX in place, but there are no unused inputs on this MUX, so we need to increase the size of this MUX from 4:1 to 8:1, which requires an extra control signal output from the control unit. The context restoration upon returning from the interrupt then required an extra 32 states of so, where the control unit is responsible once again for generating the appropriate addresses. The control unit now has 64 or so extra states and one extra control signal.

b) The assembler does not need to change. The addition of hardware did not change the instruction word width.

c) This change adds 32x8=256 bits for the extra shadow register file. Additionally, we added 64 states to the control unit's FSM, which effectively made the control unit increase from at least two bits of memory to at least seven bits of memory.

d) This change would be great if you had to switch contexts often and needed to save all the registers for that context switch. The problem is that a context switch no goes from requiring one clock cycle to requiring 32 clock cycles, which significantly increases the required time for the context switch. In other words, overall performance degrades if your system generates many interrupts and does not need to store all the registers in the register file.

Comment: Another approach to this problem is to place a separate FSM external to the control unit. When the hardware requires a context switch, the control unit would instruct the new FSM start the transfer of data from one register file to the other. The new FSM would inform the control unit when it completes the transfer. This

approach would require extra memory in the form of a counter and an external FSM, as well as some other supporting routing and selection hardware.

Device	Size	Number of Bits	New
prog_rom	1024 x 18	18432	18432
Register file	32 x 8	256	32x8x2=512
Program counter	10	10	10
Stack pointer	8	8	8
Flags (I,Z,C,shZ,shC)	5 x 1	5	5
Scratch RAM	256 x 10	2560	128x10=1280
	TOTAL	**21,271**	**21,527**

Example 20.11: Adding Two Conditional Branch Instructions

Add two new conditional branch instructions to the RAT MCU instruction set: BRVS & BRVC, which are mnemonics for "branch if overflow set" and "branch if overflow cleared", respectively. These instructions are based on the notion of overflow from specific instructions involving ALU operations. Recall that overflow results from signed binary operations where the sign-bit of the result is different from the sign-bit of the two operands when the sign bit of the operands are the same (which is not the same as a carry-out). For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) The first thing that change is the ALU, which now needs to a V output. The ALU hardware needd modifications to all of the operations to state how each operation affects the new V output. The V output is the input to a flip-flop in a similar manner to the C & Z flags, so the RAT MCU hardware needs an additional flag register. The control unit needs to change to support these two new instructions, which includes both adding a V input to the control unit and making the control unit recognize the opcodes for these two new instructions.

b) The assembler needs to change to support these two new instructions. Additionally, the assembler needs to change to support the V output for all ALU-based instructions (note that this problem did not state how V would change for any specific instruction). These two instructions are immed-type instructions, which have plenty of extra code space. Because of the extra code space, the width of the instruction words will not change.

c) This change increases memory by one extra bit to support the V flag. The instruction word remains unchanged as these instructions if into the current immed-type instruction code space.

d) This functionality would be a welcome addition to the RAT MCU's current lack of arithmetic-type functionality. There are several types of applications where this change would be useful.

Example 20.12: Adding Memory Exclusively for a Stack

You need to add a new memory to the RAT MCU that functions exclusively as a stack. In this way, the stack does not share the same physical memory as the scratch RAM. For this problem, assume the new stack size is 32 x 10 and that the scratch RAM size remains at 256 locations. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change

b. describe any changes you need to make to the RAT assembler to support this change

c. state the change in RAT memory requirements to support this change

d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) The obvious hardware change is the addition of the 32x10 RAM. But then again, since the scratch RAM now only needs to save data from the register file, we can reduce the overall size of the scratch RAM from 256x10 to 256x8. Both the scratch RAM and new stack RAM need both address and data inputs. We can remove the current scratch RAM MUX as the scratch RAM's data input is connected directly to the X output of the register file. The scratch RAM's address MUX now only needs to be a 2:1 MUX as it only needs to choose either the immediate value of a Y output of the register for an address. Thus, the new scratch RAM has one less MUX, one smaller MUX, and two less MUX control signal. The new stack RAM's data input could connect directly to the PC output to handle CALL-type instructions. The address inputs requires a 2:1 MUX to choose between the current stack pointer value and one less than the current stack pointer value. The address to the stack RAM only needs to be five bits wide to support 32 locations, so we can reduce the current stack pointer from eight to five bits in width. Lastly, we must change the control unit to deal with the required changes in control signals.

b) The assembler does not need to change as there were not changes in data transfer to the scratch RAM and the changes to data transfer to the new stack RAM are independent of the current instruction formats.

c) The changes in memory include addition of the stack RAM at 32x10 bits. The overall memory requirements decreased for the scratch RAM from 256x10 to 256x8. The stack pointer memory decreased from eight bits to five bits.

d) Have a dedicated stack would simplify the RAT MCU architecture by removing the sharing of the scratch RAM for both data and stack. While most of the stack integrity issues would remain, having a separate RAM for the stack would ensure that stack never overwrote important data in the scratch RAM.

Device	Size	Number of Bits	New
prog_rom	1024 x 18	18432	18432
Register file	32 x 8	256	256
Program counter	10	10	10
Stack pointer	8	8	5
Flags (I,Z,C,shZ,shC)	5 x 1	5	5
Scratch RAM	256 x 10	2560	256x8=2048
Stack RAM	32 x 10	-	320
	TOTAL	**21,271**	**21,076**

Example 20.13: A Useful IN Instruction Form:

Add the following instruction to the RAT MCU instruction set:
"**`IN (r3),PORT_ID`**", where r3 is the destination operand that holds the address in scratch RAM of where you want to place the data and "PORT_ID" is the same as in the current IN instruction. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change

b. describe any changes you need to make to the RAT assembler to support this change

c. state the change in RAT memory requirements to support this change

d. describe why this change could be useful

Solution: This problem asks you to add something to the RAT MCU, which means there is a possibility that the width of the instruction word will increase.

a) The main hardware issue with making this change is that any added hardware needs to connect with the current scratch RAM. The current data input the scratch RAM has a 2:1 MUX so that it can choose between either register data or immediate data. This MUX needs to increase to a 4:1 MUX in order to connect the IN_PORT value to data input of the scratch RAM. This change requires the select signals for this MUX to be two bits rather than the current one bit, which requires changing the control unit.

   As for the address to the scratch RAM, there are a few ways to do this. The problem is that this is reg-immed-type instruction. That being the case, the reg value is associated with the X input and subsequently the DX_OUT signal. The problem here is the DX_OUT signal does not currently connect with the address input to the scratch RAM. One approach to handle this is to change the 4:1 address MUX to the scratch RAM to an 8:1 MUX and add the DX_OUT signal to the MUX. This approach requires the width of the select signal to the MUX to increase to three bits from the current two. Another approach to this change is to add a 2:1 MUX in front of the ADRY input of the register file; the MUX inputs are the ADRX and ADRY values. For this instruction, the control unit selects the standard ADRX bits (IR(12:8)) as the input to ADRY; for all other instructions, the ADRY input uses the standard IR(7:3) bits. In this way, this instruction then chooses to use the ADRX value to input to the ADRY input in the case when this new instruction executes. This works nicely because the DY_OUT value is currently routed to the address input MUX of the scratch RAM. There are other ways to do this also.

   The control unit needs to change to recognize this instruction and send out the appropriate control signals. The control unit with also need two extra control signals that it could be used to control various MUXes based on the approach used to route the data and address signals supporting this instruction.

b) The assembler needs to be made aware of this new instruction. This includes assigning the new instruction an op-code. This instruction is a reg-immed-type instruction, and there is room (although not much) in the reg-immed instruction space to assign this instruction a new op-code without extending the current 18-bit length of the RAT MCU instructions.

c) This instruction causes no changes to the current RAT MCU memory requirements. "Adding the instruction to the control is essentially modifying an internal decoder of the control unit. Recall that decoders are generally combinatorial circuits. We did not alter the number of states in control unit by adding this instruction so there was no increase in memory requirements.

d) Currently, all data input from the outside world must be place in a register. In other words, input operations are register-based. This instruction allows you to place input data directly into the scratch RAM. As with all scratch RAM-based operations, the scratch RAM requires an address of where to read or write the data. For this new instruction, the destination operand provides the address using a register-indirect for as is currently done with one form of the LD and ST instructions. Thus, the advantage of this instruction is that it saves instructions (thus code space in instruction memory) for input data that needs to writing to the scratch RAM. Table 20.1 show a comparison of the old and new functionality.

Current RAT MCU Functionality	New Instruction Functionality
`IN r4,0x45`	`IN (r3),0x45`
`ST r4,(r3)`	

Table 20.1: Comparison of new instruction functionality.

Example 20.14: Adding a HALT Instruction

I want to provide the RAT hardware with the ability to pause program execution. Many CPUs contain a "HALT" instruction, so I want the RAT to have one also. Describe the changes I would need to make to the associated RAT hardware and the RAT assembler in order to implement this instruction and any other instruction I would need as a result of executing a "HALT" instruction. Also, estimate the increase or decrease in the overall bits of storage in the RAT MCU caused by this change. For this problem, do the following:

a. describe the changes you need to make to the RAT hardware to support this change
b. describe any changes you need to make to the RAT assembler to support this change
c. state the change in RAT memory requirements to support this change
d. describe why this change could be useful

Solution: The RAT MCU is always doing something; the way the RAT was designed, it never really needs to stop doing anything. HALT-type instructions are useful in many ways, the ways are not worth going into here. The most obvious advantage would be to reduce power consumption by stopping the RAT MCU from doing anything. Once again, this is one of those problems with many possible solutions; there is nothing special about this solution, so hopefully you can come up with a better one yourself.

Implementing this instruction would require the Control Unit to be modified in order to recognize this instruction and send out the appropriate control signals. Adding an instruction does not change the number of states in the control unit will thus not change memory requirements. We would implement this instruction by causing the RAT MCU to go into a "HALT" state if this instruction were executed. This could possibly cause an increase in the memory associated with the control unit as we're officially adding a state.

The main issue with this instruction is how you would restart the RAT MCU once you executed a HALT instruction. The question is rather misleading on this issue as it suggests that you must add another instruction to "START" the CPU. But, if the RAT MCU is HALTed, you won't be able to execute an instruction. The only solution is to have some external signal "unhalt" the RAT MCU. In this case, the HALT instruction would cause the RAT Control Unit to go into a "HALT" state; in this case, some external signal, strangely similar to an interrupt, would be required to get the RAT MCU doing something meaningful again by leaving the HALT state. This signal could be anything, such as a user button-press or something similar. Keep in mind in real life, CPUs do all they can to turn themselves off if they are not being used; these are referred to as

power-saving modes. But, they need to quickly wake up when something important needs the CPUs computational abilities.

The assembler would need to change in order to support this instruction. The changes would be minor as this instruction would be of none-type; implementing this instruction would require assigning an opcode. As with all none-type instructions, there is no field code associated with the instruction. The instruction word length would not need to change as there is plenty of code space for none-type instructions.

20.3 Chapter Summary

- This chapter presented a bunch of possible modifications to the RAT MCU. The learning element here is comes by understanding the solutions.
- Any changes to hardware may affect the overall instruction word size. You must reference the five RAT MCU's instruction formats when working with hardware changes such as these. A complete understanding of the instruction formats is vital when working with these problems.
- The main issue to consider with "modifications" is that if you "add" something to the hardware, there is a distinct possibility that you may need to extend the length of the RAT MCU's instruction word. This idea includes increasing the size of an existing hardware module. Conversely, if you "remove" something from the RAT MCU hardare, there is a distinct possibility that you can decrease the size of the RAT MCU's instruction word. This idea includes reducing the size of an existing hardware module.

20.4 Chapter Exercises

1) Implement the following instruction: “INOUT”. This instruction has the form: “INOUT r0,0x23” where the source operand is considered a port_id. This instruction will simultaneously input a value and write it to r0 (the destination operand) and also output the value to the port_id (the source operand).

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

2) Since the C flag is not being used on the AND-type instructions, it indicate the parity of the result of the AND operation. For this problem, do the following:

   - describe the changes I would need to make to the RAT hardware for this modification
   - describe any changes I would need to make to the RAT assembler to support this modification
   - estimate the change in RAT memory requirements from this modification
   - show how you would use the new AND instruction to generate the parity of the value in register r8
   - comment on whether you feel your approach would be easier than simply making a “P” flag

3) The RAT MCU’s scratch RAM is 10-bits wide, but only eight of those bits are used by the register. Implement an instruction: “WBITS” that writes the two unsed bits to a register. The instruction will have this form “WBITS r3,0x33”, where the two bits come from scratch RAM address 0x33 and are written to the lower two bits of r3 (the unused bits in r3 are automatically cleared).

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

4) Implement the following instruction: “IN 0x37,0x23” where the source operand (right operand) is considered a port_id and the destination operand is an address used to index scratch RAM. This instruction inputs the value from the input port to the given scratch RAM address.

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

5) Implement the following instruction: "ROLST r1,(r2)" where the destination operand (r1) is rotated left and the result is stored in the scratch RAM location referenced by the value in the source operand (r2).

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

6) Implement the following instruction: "OUT 0x37,0x23" where the source operand is considered a port_id and the destination operand is an address used to index scratch RAM. This instruction outputs the value in scratch RAM to the given port_id.

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

7) The RAT MCU "OUT" instruction only outputs from a register to the RAT's output lines. For this problem, I want to implement an instruction that outputs directly from a memory address to the RAT's output lines. An example of this instruction would be: "**`OUT (r3),PORT_ID`**", where register r3 contains the address of the memory data to output and PORT_ID is the port_ID as it is in the current OUT instruction. For this problem, do the following:

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

8) Implement a set of "conditional CALL" instructions: CALLZ, CALLNZ, CALLC, CALLNC. These instructions make a subroutine call based on the state of the condition flags.

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

9) Implement a set of "conditional RET" instructions: RETZ, RETNZ, RETC, RETNC. These instructions make a subroutine call based on the state of the condition flags.

   - describe the changes I would need to make to the RAT hardware for this instruction
   - describe any changes I would need to make to the RAT assembler to support this instruction
   - estimate the change in RAT memory requirements from the changes
   - describe why such an instruction would potentially be useful

10) Implement an instruction that automatically increments the index register for indirect LD instructions. For example, when I write "**`LD r0,(r1+)`**", the value from address value in r1 is loaded from memory into r0 and the value in r1 is automatically incremented as part of this operation. For this problem, do the following:

- describe the changes I would need to make to the RAT hardware for this instruction
- describe any changes I would need to make to the RAT assembler to support this instruction
- estimate the change in RAT memory requirements from your change
- describe an area where such an instruction would be potentially useful

11) I want to implement what is typically referred to as an "indexed addressing mode". What this means is that I want to be able to issue this instruction: " LD r0,(xr1)". What this instruction does is adds the value in r1 and r31 to use as an index into scratch RAM. You can use any register as the source operand, but the presence of the "x" prefix means that you also are adding the contents of r31 to generate the absolute address value.

- describe the changes I would need to make to the RAT hardware for this instruction
- describe any changes I would need to make to the RAT assembler to support this instruction
- estimate the change in RAT memory requirements from the changes
- describe why such an instruction would potentially be useful

12) I want to implement the following instructions: "SRSWP" which stands for "scratch RAM swap". This instruction has the form "SRSWP r1,(r2)", which means it swaps the value in r1 with the contents of the memory location indicated by the value in r2.

- describe the changes I would need to make to the RAT hardware for this instruction
- describe any changes I would need to make to the RAT assembler to support this instruction
- estimate the change in RAT memory requirements from the changes
- describe why such an instruction would potentially be useful

13) You must modify the RAT MCU architecture by changing the size of the register file from 32x8 to 8x8 and the prog_rom from 1k instructions to 4k instructions. For this problem, describe the following:

a) changes you need to make to the RAT hardware
b) changes you need to make to the RAT assembler
c) changes in RAT MCU memory requirements
d) why this modification would be quite useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
Scratch RAM	256 x 10	2560
	TOTAL	**21,271**

14) Modify the RAT MCU so that it can read 10-bit values from the scratch RAM and place them in the register file. The approach is to issue an instruction that allows for all subsequent LD instructions to either load the upper or lower eight bits to the register file. The new instructions are listed below with further description.

```
LD_HI      ; causes all subsequent LD instructions to load the 8 MSBs of
           ;  10-bit scratch RAM value to regfile
LD_LO      ; causes all subsequent LD instructions to load the 8 LSBs of
           ;  10-bit scratch RAM value to regfile
```

a) changes you need to make to the RAT hardware
b) changes you need to make to the RAT assembler
c) changes in RAT MCU memory requirements
d) why this modification would be useful

Device	Size	# bits
prog_rom	1024 x 18	18432
register file	32 x 8	256
program counter	10	10
stack pointer	8	8
flags(I,Z,C,shZ,shC)	5 x 1	5
scratch RAM	256 x 10	2560
	TOTAL	**21,271**

21 External Device Interfacing

21.1 Introduction

Simple microcontrollers such as the RAT MCU don't have the capability making meaningful circuits on their own. Generally speaking, in order to make the RAT MCU do something meaningful, you have to interface the RAT MCU with various external devices. While there are microcontrollers out there that have fun and happening devices included in the MCU, the RAT MCU is relatively simple in comparison to the features contained in modern microcontroller.

Interfacing microcontrollers to external devices, or peripherals, is the name of the modern embedded systems design game. The general idea is that you must familiarize themselves the device you want to interface with in order to properly program the RAT MCU.

Main Chapter Topics

- **DEVICE INTERFACE OVERVIEW:** This chapter presents a model for designing external device interfaces.
- **LOW-LEVEL MEMORY INTERFACE:** This chapter provides an example of interfacing to an external memory device.
- **LOW-LEVEL ADC INTERFACE:** This chapter provides an example of interfacing the RAT MCU to an external analog-to-digital converter.

Why This Chapter is Important

This chapter is important because it provides some low-level details of interfacing to common electronic devices.

21.2 Designing Device Interfaces

There is a world full of digital devices out there that are waiting to be controlled by microcontrollers such as the RAT. Keep in mind that the RAT is a versatile digital device in that you can easily reprogram the hardware in order to control just about any other digital device. This is good news because there are about a bajillion different devices out there waiting for you to include them in your next design. The good news is that we can describe the interface design process in generic terms. After you've identified a device that you want to control, there are three basic steps you can take to actually implement your design.

1. Define device interface requirements: Read through the datasheet for the peripheral device in order to define which signal on the device will be used to communicate with the microcontroller. These signals are essentially control and status signals: the control signals control the peripheral device and the status signals communicate status information from the peripheral device to the microcontroller.

2. Design required interface hardware: The RAT MCU is a generic design so you'll probably need to design special hardware to facilitate the interface. This is basic hardware design that is similar to the previous design that we've worked with.

3. Design required firmware to interface with hardware: We often refer to this step to as writing the *device driver*. In this context, a device driver is nothing more than some software (or firmware) that controls some hardware. The term "device driver" is simply a fancy name for writing some code that does something useful in a real embedded system.

21.3 Interfacing RAT to an EEPROM

One of the drawbacks of RAT is that it does not have a lot of generic memory that the programmer can use for storing items such as intermediate results. We can view this lack of data memory as a severe limitation in the basic RAT MCU. Because the RAT MCU has primitive and generic I/O capabilities, it is straightforward to have the RAT interface to external peripheral devices such as memory. This section details the hardware and firmware requirements of interfacing a generic EPROM memory to the RATMCU. We have removed some of the details to protect the innocent but most of the fun and interesting stuff remains.

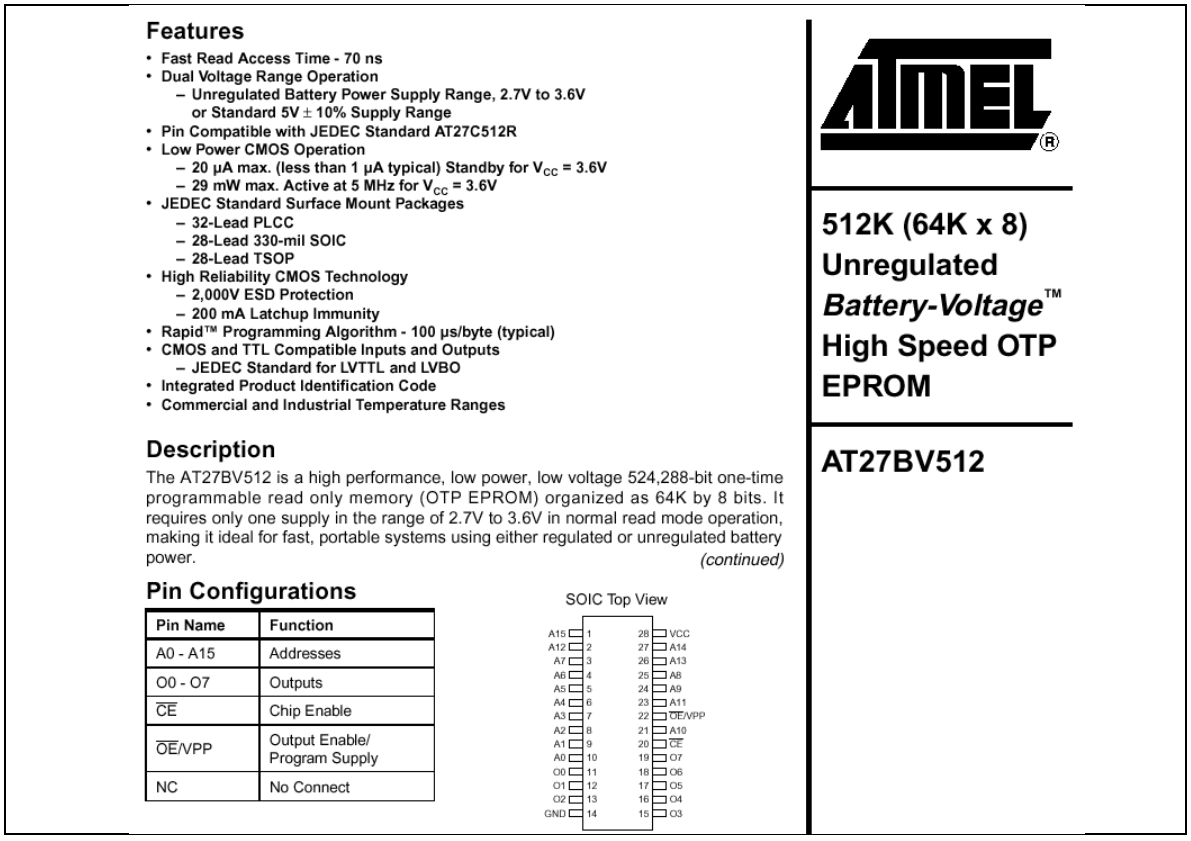
The device we'll be working with is a 64K x 8 EPROM. We'll assume that RAT only needs to read from the device for the sake of simplicity (thus, it does not have to write to the device). As you will see, this device is an OTP (one time programmable) device. In this way, we can assume that some other device has previously written to the memory. Although having a read-only capability does not make the RAT seem all that more versatile, the intent here is to present an overview of the interface requirements and procedure for a relatively generic device.

Figure 21-1 shows most of the first page of the EPROM datasheet. The front page in any datasheet typically provides a lot of useful information because the manufacture is attempting to lure you into using their part. Here are the important features of Figure 21-1.

- The device can store 64K bytes of information. Since each of these bytes of information can be uniquely addressed, the device must necessarily have 16 address lines.

- Since the output width of the stored data is eight bits, the device contains eight data output lines.

- There are two control lines for the device: CE and OE. Both of these signals are active low. The CE is a *chip enable* while the OE is an *output enable* (the OE acronym is generally associated with reading data as opposed to writing data).



Features

- Fast Read Access Time - 70 ns
- Dual Voltage Range Operation
  - Unregulated Battery Power Supply Range, 2.7V to 3.6V or Standard 5V ± 10% Supply Range
- Pin Compatible with JEDEC Standard AT27C512R
- Low Power CMOS Operation
  - 20 µA max. (less than 1 µA typical) Standby for V_{CC} = 3.6V
  - 29 mW max. Active at 5 MHz for V_{CC} = 3.6V
- JEDEC Standard Surface Mount Packages
  - 32-Lead PLCC
  - 28-Lead 330-mil SOIC
  - 28-Lead TSOP
- High Reliability CMOS Technology
  - 2,000V ESD Protection
  - 200 mA Latchup Immunity
- Rapid™ Programming Algorithm - 100 µs/byte (typical)
- CMOS and TTL Compatible Inputs and Outputs
  - JEDEC Standard for LVTTL and LVBO
- Integrated Product Identification Code
- Commercial and Industrial Temperature Ranges

Description

The AT27BV512 is a high performance, low power, low voltage 524,288-bit one-time programmable read only memory (OTP EPROM) organized as 64K by 8 bits. It requires only one supply in the range of 2.7V to 3.6V in normal read mode operation, making it ideal for fast, portable systems using either regulated or unregulated battery power. *(continued)*

Pin Configurations

Pin Name	Function
A0 - A15	Addresses
O0 - O7	Outputs
$\overline{CE}$	Chip Enable
$\overline{OE}$/VPP	Output Enable/ Program Supply
NC	No Connect

SOIC Top View

ATMEL

512K (64K x 8) Unregulated *Battery-Voltage*™ High Speed OTP EPROM

AT27BV512

Figure 21-1: Most of the first page of the datasheet.

Datasheets typically provide all the known information regarding the operation of the device, and sometimes, it actually reflects how the device actually works[1]. The information they contain is usually correct but don't bet the farm on it. There are often mistakes in manufacturer's data sheets so you should always remain open to this prospect. For this discussion, we'll not concern ourselves with the electrical characteristics and we'll assume these details have been taken care of for us. To get a feel for some of the electrical requirements, Figure 21-2 shows a few snippets from that page in the datasheet. It's somewhat dry stuff but there are actually people who find such information exciting. I don't happen to be one of them.

For this discussion, we'll be focusing our interest on the timing characteristics required to read data from the device. This information is usually found in the datasheet in the form of an annotated timing diagram. Figure 21-3 shows this information from the datasheet for the device in question.

[1] Bad attempt at humor. The spec was probably written by some overwork engineer trying to meet a deadline.

DC and AC Operating Conditions for Read Operation

		AT27BV512			
		-70	-90	-12	-15
Operating Temperature (Case)	Com.	0°C - 70°C	0°C - 70°C	0°C - 70°C	0°C - 70°C
	Ind.	-40°C - 85°C	-40°C - 85°C	-40°C - 85°C	-40°C - 85°C
V_{CC} Power Supply		2.7V to 3.6V	2.7V to 3.6V	2.7V to 3.6V	2.7V to 3.6V
		5V ± 10%	5V ± 10%	5V ± 10%	5V ± 10%

DC and Operating Characteristics for Read Operation

Symbol	Parameter	Condition	Min	Max	Units
V_{CC} = 2.7V to 3.6V					
I_{LI}	Input Load Current	V_{IN} = 0V to V_{CC}		±1	µA
I_{LO}	Output Leakage Current	V_{OUT} = 0V to V_{CC}		±5	µA

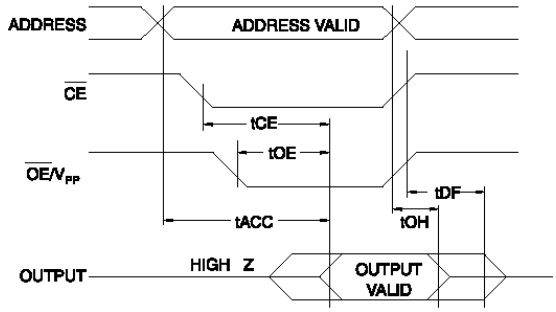
Figure 21-2: A few of the endless electrical characteristics.

AC Characteristics for Read Operation

V_{CC} = 2.7V to 3.6V and 4.5V to 5.5V

			AT27BV512								
			-70		-90		-12		-15		
Symbol	Parameter	Condition	Min	Max	Min	Max	Min	Max	Min	Max	Units
t_{ACC}[3]	Address to Output Delay	$\overline{CE}$ = $\overline{OE}/V_{PP}$ = V_{IL}		70		90		120		150	ns
t_{CE}[2]	$\overline{CE}$ to Output Delay	$\overline{OE}/V_{PP}$ = V_{IL}		70		90		120		150	ns
t_{OE}[2][3]	$\overline{OE}/V_{PP}$ to Output Delay	$\overline{CE}$ = V_{IL}		50		50		50		60	ns
t_{DF}[4][5]	$\overline{OE}/V_{PP}$ or $\overline{CE}$ High to Output Float, whichever occurred first			40		40		40		50	ns
t_{OH}	Output Hold from Address, $\overline{CE}$ or $\overline{OE}/V_{PP}$, whichever occurred first		0		0		0		0		ns

AC Waveforms for Read Operation[1]



Notes:
1. Timing measurement references are 0.8V and 2.0V. Input AC drive levels are 0.45V and 2.4V, unless otherwise specified.
2. $\overline{OE}/V_{PP}$ may be delayed up to t_{CE}-t_{OE} after the falling edge of $\overline{CE}$ without impact on t_{CE}.
3. $\overline{OE}/V_{PP}$ may be delayed up to t_{ACC}-t_{OE} after the address is valid without impact on t_{ACC}.
4. This parameter is only sampled and is not 100% tested.
5. Output float is defined as the point when data is no longer driven.
6. When reading a 27BV256, a 0.1 µF capacitor is required across V_{CC} and ground to suppress spurious voltage transients.

Figure 21-3: The all-important timing diagram associated with the EPROM.

The general thought here is that if I can make the RAT MCU mimic the control signals with their timing requirements shown in Figure 21-3, I'll be able to successfully interface (get data from) the device. The reality is that the RAT in its raw form is not configured to directly talk to such a device. But with the addition of proper hardware and appropriate software, you'll be able to communicate with the device. This general approach is often referred to as being *bit-banging*. This directly refers to the fact that you'll need to use the RAT MCU to "synthesize" the proper signals to control the peripheral device in question.

The RAT, with its generic interface, will not automatically interface with anything. You need to create the specifics of the interface yourself. Once this is done, you need to synthesize the required control signals by using the I/O capabilities of the RAT. In essence, this means that you need to make output bits go high and low under program control in order to mimic the I/O requirements specified by the device. Here are some of the highlights (signals that we care about) of the EPROM device:

- One of the most important characteristics regarding any memory device is the access time. In this case, we're interested in the read access time since we've assumed that the device was written to (data was placed in it) by some other device. The access time refers to the amount of time required for valid data to appear on the device outputs once the address lines are valid and the proper control signals are exercised.

- Note that there are four different access times listed for this particular device. The reality is that not all memory devices are created equal. In this case, the -xx numbers refer to the access time. The devices with the shorter access times are generally more expensive parts.

- Figure 21-3 lists two notes that are important to us: notes two and three. The idea is that if we can reproduce the timing diagram shown in Figure 21-3 we'll be able to communicate with the EPROM. But, the two notes specify that we don't need to generate the timing diagram exactly. Note two effectively states that the delay from the OE signal going low after the CE signal goes low is effectively zero. Note three effectively states that the time the OE signal goes low does not affect the memory access time. These two notes together indicate that both the CE and OE signals can be brought low at the same time without affecting the access time.

21.3.1 Interfacing RAT to a Memory Module

We're to the point where we'll assume we know how the memory operates and we're now ready to generate the hardware used to interface with the device. Figure 21-4 shows the signals on the memory that we're interested in for the interface requirements Figure 21-5 shows the RAT block diagram. Both of these figures are important in the final interface circuit shown in Figure 21-6.

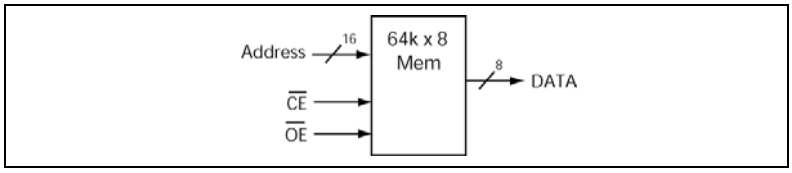


Figure 21-4: The signals of digital interest on the memory.

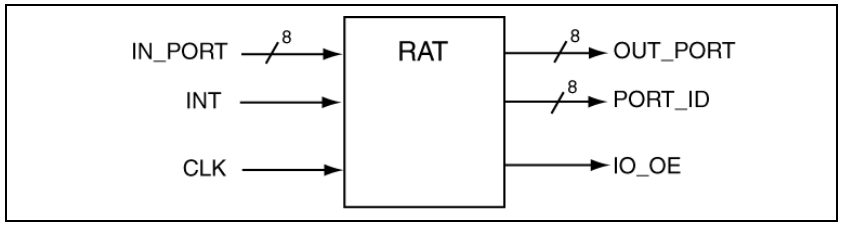


Figure 21-5: RAT black box diagram.

Example 21.1

Write a RAT assembly language subroutine that could be used to interface with the previously described EPROM and using the circuit shown in Figure 21-6. The address of the desired data will be sent in registers r10 (high address) and r11 (low address). The associated data should be placed in register r14. For this interface problem, assume that the -15 EPROM part is the one used in the design. Also assume that the RAT system clock is running at 100MHz.

Solution: Figure 21-6 shows the critical interfacing of the circuit. The WR_Strobe signal should actually be IO_STRB on the RAT MCU. This is not the only way to interface with the memory; there are many possible combinations. Another thing to keep in mind before we describe this interface is that the signal assignments on many parts of this circuit are arbitrary. The thought here is that someone may have set up this circuit for you and it is now your job to interpret this circuit so you can write the firmware that will control it.

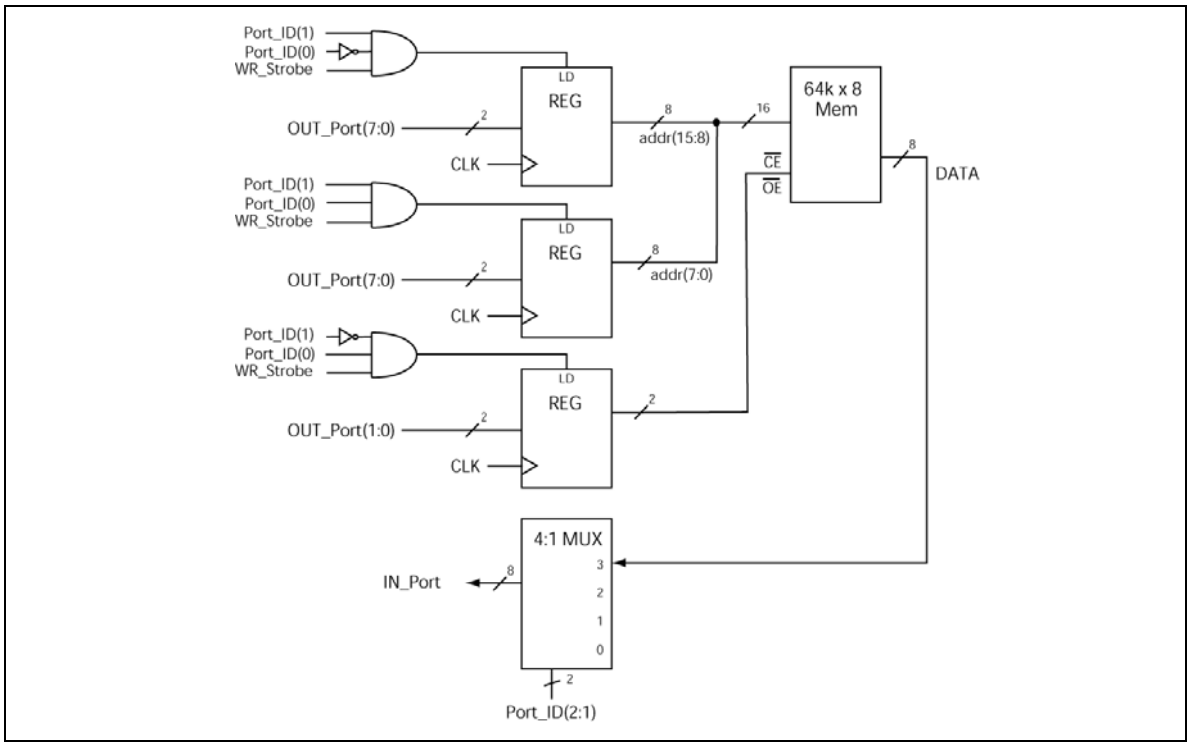


Figure 21-6: The final interface circuit including the memory device.

- The CE and OE signals are registered as are the address lines. The top register holds the data for the higher eight address lines (15:8) while the lower register holds the data for the lower eight address lines (7:0). Loading of the registers is controlled by the WR_Strobe and Port_ID(1:0) signal from the RAT. Keep in mind that the Port_ID(1:0) are two of the signals on the Port_ID output of the RAT as is shown in Figure 18-18. Assignment of Port_ID(1:0) to control these registers is arbitrary. The input of the register is OUT_Port signals from the RAT.

- There are eight output signals from the memory representing the data store in the memory. Although the memory output could have been connected directly to the RAT IN_Port inputs, they are instead fed to a 4:1 MUX and controlled by Port_ID signals 2:1. The Port_ID(2:1) assignment is once again arbitrary.

- Table 21.1 shows the addresses used to access the hardware shown in Figure 21-6.

Constant Specifier	Value	Purpose
MEM_CTRL	0x01	Used to write CE and OE values to memory
MEM_ADDR_HI	0x02	Used to write high address to memory: addr(15:8)
MEM_ADDR_LO	0x03	Used to write low address to memory: addr(7:0)
MEM_DATA	0x06	Used to read data from memory

Table 21.1: Port addresses for the hardware specified in Figure 21-6.

21.3.2 Generating the Flowchart

The first step in any program you write should be to generate a flowchart. For this problem, generating the flowchart allows you to find out whether you really understand the problem or not. If you don't really understand the problem, sometimes a flowchart maps you a path to successfully implementing the code. If you do know what is going on, the flowchart helps make your code neat, organized and readable. Figure 21-7 Shows the flowchart associated with this subroutine. Note that this flowchart is relatively high-level in that it does not indicate low-level details of address reading or delay implementation.

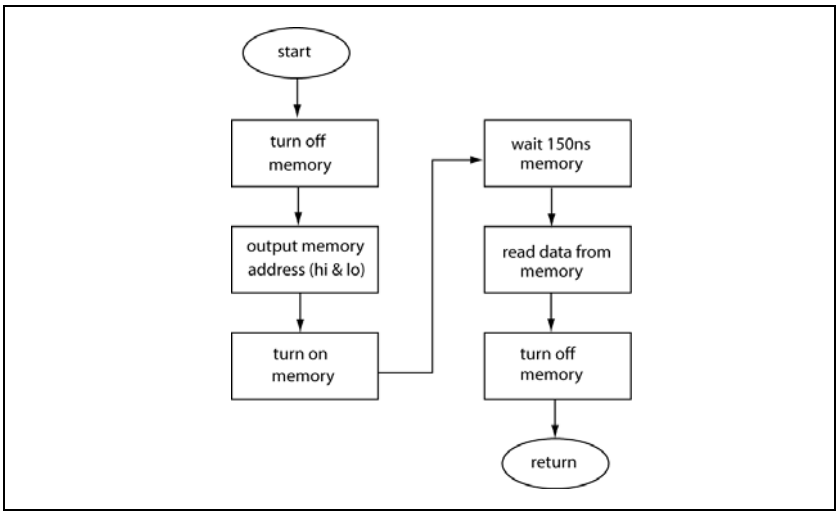


Figure 21-7: The flowchart associated with the example problem.

21.3.3 Writing the Firmware

Now that the hardware is setup, we're ready to write the firmware. The problem at hand here is to write a subroutine that reads a byte of data from memory. The address of the desired data will be sent in registers r10 (high address) and r11 (low address). We place the associated data in register r14. For this interface problem, assume that the -15 EPROM part is the one used in the design. Also, assume that the RAT system clock is running at 100MHz.

```
;-------------------------------------------------------------------------
;- Assembler directives for control and data addressing
;-------------------------------------------------------------------------
.EQU  MEM_CTRL    = 0x01   ; port address for memory control signals (CE,OE)
.EQU  MEM_ADDR_HI = 0x02   ; port address for hi memory address signals
.EQU  MEM_ADDR_LO = 0x03   ; port address for lo memory address signals
.EQU  MEM_DATA    = 0x06   ; port address for memory DATA signals
;-------------------------------------------------------------------------

;-------------------------------------------------------------------------
;- Subroutine mem_read
;-  This subroutine reads from EPROM memory device. The address to read
;-  from is sent in registers r10 and r11 (hi and lo address). The resulting
;-  data should be placed in register r14.
;-
;- Registers Used: r0
;-------------------------------------------------------------------
mem_read:
         MOV     r0,0xFF           ; make sure memory is off (this step
         OUT     r0,MEM_CTRL       ;  starts the process)
                                   ;
         OUT     r10,MEM_ADDR_HI   ; register the high address
         OUT     r11,MEM_ADDR_LO   ; register the low address
                                   ;
         MOV     r0,0x00           ; turn on memory device
         OUT     r0,MEM_CTRL       ;
                                   ;
         MOV     r0,0x04           ; load delay value: we must wait 150ns for
wait:    AND     r0,r0             ;  data to be valid (at 100MHz, 20ns per
         BRNE    wait              ;  instruction, wait four iterations)
                                   ;
         IN      r14,MEM_DATA      ; get the data from memory.
                                   ;
         MOV     r0,0xFF           ; turn off the memory.
         OUT     r0,MEM_CTRL       ;
                                   ;
         RET                       ; pass control back to calling routine.
;-------------------------------------------------------------------------
```

Figure 21-8: RAT assembly code that implements a read of the memory.

21.4 Interfacing RAT to an Analog-to-Digital Converter

The ability for a computer to input and output information is what makes computers useful. The added benefit is that they do it relatively fast. The really cool thing is that the concepts involved in doing such interfacing are straightforward enough so any engineering student can design and implement microcontroller-based systems with relative ease (can you say “Arduino”?). A tendency in some engineering students is to consider digital design a pursuit that requires extensive programming in order to accomplish anything useful. The thought of programming something most likely elicits bad memories from classes taught by computer science instructors who got more of a thrill by flunking you rather than teaching you. The reality is that digital system design at the microcontroller level is straightforward for just about anyone who has the courage not to be intimidated by the concepts, the instruction sets, the programmers models, and most of all, the associated datasheets.

This section is another example of the steps that would be required in order to interface the RAT microcontroller with a peripheral device. In this context, a peripheral device is some device external to RAT that provides some sort of information regarding the outside world (generally a sensor of some sort). As you probably remember from the previous section, the RAT design is generic so it can interface to just about any digital device out there. This section describes the interfacing requirements for a relatively generic analog-to-digital converter (ADC).

Despite the fact that this section is somewhat specific to an ADC, try to keep in touch with the fact that steps involved are somewhat generic and can be used as a road map to interfacing any device. Keep in mind that

there are only three basic steps in this process once you've identified the device you intend to interface with. Here are those steps again:

1. Define device interface requirements: Read through the datasheet for the peripheral device in order to define which signal on the device will be used to communicate with the microcontroller. These signals are essentially control and status signals: the control signals control the peripheral device and the status signals communicate status information from the peripheral device to the microcontroller.

2. Design required interface hardware: The RAT MCU is a generic design so you'll probably need to design special hardware to facilitate the interface. This is basic hardware design that is similar to the previous design that we've worked with.

3. Design required firmware to interface with hardware: We often refer to this step to as writing the *device driver*. In this context, a device driver is nothing more than some software (or firmware) that controls some hardware. The term "device driver" is simply a fancy name for writing some code that does something useful in a real embedded system.

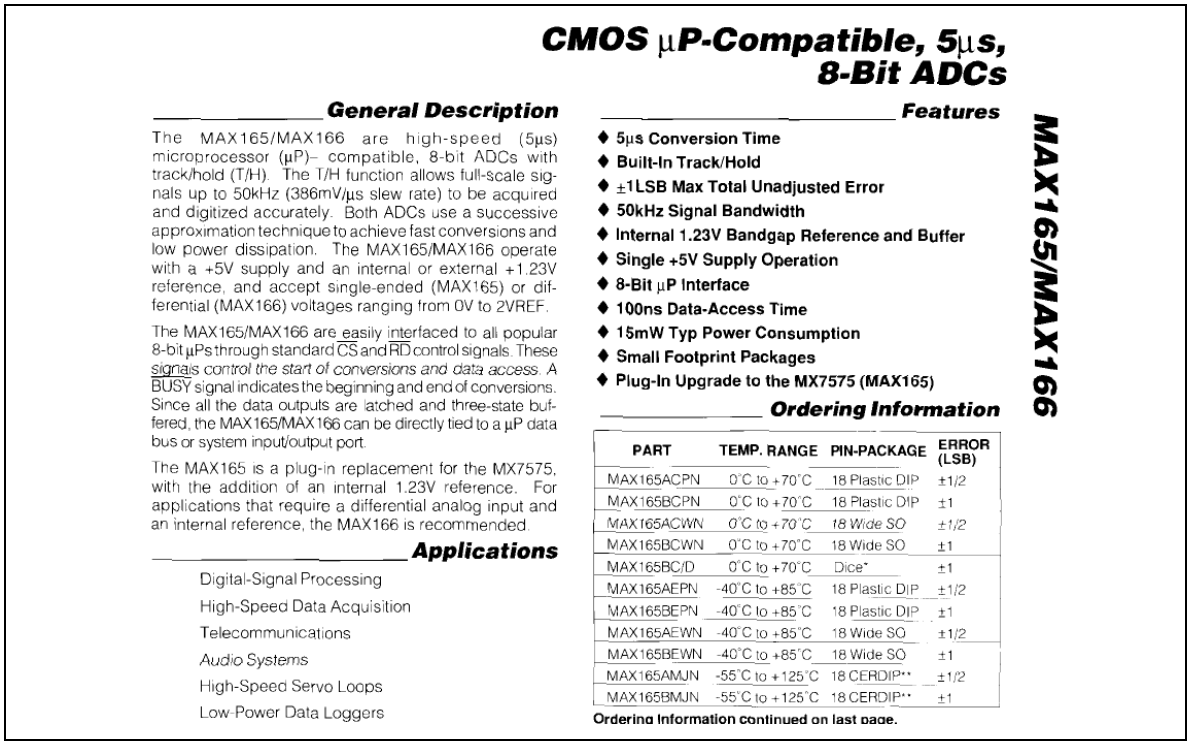
21.4.1 ADC Specifics

An ADC is probably the most commonly used peripheral device in digital design. Keep in mind that although we live in an analog world[2], computers are essentially digital devices. For this reason, ADCs are the primary mechanism that computers use to interface with the real world. The ADC inputs an analog voltage, converts this voltage to a digital number with a predetermined number of bits, and makes the digital representation of the number available to the outside world. The number of bits associated with the ADC is pre-set and is a function of the particular device that you're using. ADCs inherently have both digital and analog control inputs but we'll only be interested in the digital stuff for this discussion.

This discussion uses a generic 8-bit ADC, where the digital output value is an 8-bit binary number that is represents digital version of the analog input voltage. The associated datasheet is about twenty pages long but we'll only be looking at a few of the more important items that we'll use to interface to the device. For the remainder of this discussion, you can assume that the device was previously setup and properly configured. The setup and configuration is not overly complicated but it's outside of the realm of this discussion. Be sure to check out the datasheet for the full story.

Figure 21-1 shows most of the first page of the ADC datasheet. The front page in any datasheet typically provides a lot of useful information because the manufacture is attempting to lure you into using their part. The second paragraph describes the important features of this figure relative to this discussion; these are the control signals (RD, CS, and BUSY) and the "5µs" figure. What these roughly means is that our interface requires three signal to control the device and an analog-to-digital conversion requires 5µs ($5x10^{-6}$ seconds) to complete. These are important values and we'll be working with them shortly.

[2] In this context, analog refers to *continuous* as opposed to *discrete*. For example, the ratio 1/3 is nicely defined in the continuous world but we can only approximate this number in the digital world. This approximation is due to the limited number of *bits* that we can use to represent the number in a computer setting (it's the limited register size thing all over again).



CMOS µP-Compatible, 5µs, 8-Bit ADCs

MAX165/MAX166

General Description

The MAX165/MAX166 are high-speed (5µs) microprocessor (µP)–compatible, 8-bit ADCs with track/hold (T/H). The T/H function allows full-scale signals up to 50kHz (386mV/µs slew rate) to be acquired and digitized accurately. Both ADCs use a successive approximation technique to achieve fast conversions and low power dissipation. The MAX165/MAX166 operate with a +5V supply and an internal or external +1.23V reference, and accept single-ended (MAX165) or differential (MAX166) voltages ranging from 0V to 2VREF.

The MAX165/MAX166 are easily interfaced to all popular 8-bit µPs through standard $\overline{CS}$ and $\overline{RD}$ control signals. These signals control the start of conversions and data access. A $\overline{BUSY}$ signal indicates the beginning and end of conversions. Since all the data outputs are latched and three-state buffered, the MAX165/MAX166 can be directly tied to a µP data bus or system input/output port.

The MAX165 is a plug-in replacement for the MX7575, with the addition of an internal 1.23V reference. For applications that require a differential analog input and an internal reference, the MAX166 is recommended.

Applications

Digital-Signal Processing
High-Speed Data Acquisition
Telecommunications
Audio Systems
High-Speed Servo Loops
Low-Power Data Loggers

Features

- ♦ 5µs Conversion Time
- ♦ Built-In Track/Hold
- ♦ ±1LSB Max Total Unadjusted Error
- ♦ 50kHz Signal Bandwidth
- ♦ Internal 1.23V Bandgap Reference and Buffer
- ♦ Single +5V Supply Operation
- ♦ 8-Bit µP Interface
- ♦ 100ns Data-Access Time
- ♦ 15mW Typ Power Consumption
- ♦ Small Footprint Packages
- ♦ Plug-In Upgrade to the MX7575 (MAX165)

Ordering Information

PART	TEMP. RANGE	PIN-PACKAGE	ERROR (LSB)
MAX165ACPN	0°C to +70°C	18 Plastic DIP	±1/2
MAX165BCPN	0°C to +70°C	18 Plastic DIP	±1
MAX165ACWN	0°C to +70°C	18 Wide SO	±1/2
MAX165BCWN	0°C to +70°C	18 Wide SO	±1
MAX165BC/D	0°C to +70°C	Dice*	±1
MAX165AEPN	-40°C to +85°C	18 Plastic DIP	±1/2
MAX165BEPN	-40°C to +85°C	18 Plastic DIP	±1
MAX165AEWN	-40°C to +85°C	18 Wide SO	±1/2
MAX165BEWN	-40°C to +85°C	18 Wide SO	±1
MAX165AMJN	-55°C to +125°C	18 CERDIP**	±1/2
MAX165BMJN	-55°C to +125°C	18 CERDIP**	±1

Ordering Information continued on last page.

Figure 21-9: Most of the first page of the datasheet.

Figure 21-10 shows the pinout and functional block diagram of the ADC. The reality is that we're pretending that someone else has set up the ADC for us. Keep in mind that the ADC is a real device that contains real circuitry. The particular circuitry used in the ACD comes in different flavors and contains different characteristics that are generally important to know and understand for your particular design. Some of the signal names on the pinout diagram involve configuration on the analog end of the device and we'll be skipping over those for now. Once again, Figure 21-2 shows some of the parameters regarding the interior workings of the ADC. These type of characteristics usually run on for a few pages. For this discussion, we're more interested in the timing characteristics shown in Figure 21-12.

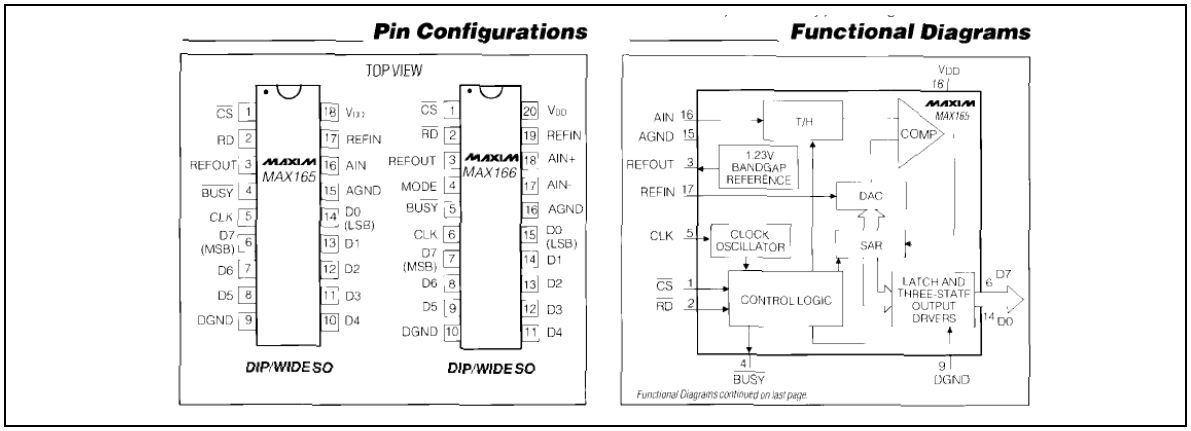


Figure 21-10: The pinout and block diagram of the ADC.

ELECTRICAL CHARACTERISTICS

(V_{DD} = +5V, REFIN = +1.23V, AGND = DGND = 0V, AIN- = 0V (MAX166), f_{CLK} = 4MHz external, T_A = T_{MIN} to T_{MAX}, unless otherwise noted.)

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNITS
ACCURACY						
Resolution			8			Bits
Total Unadjusted Error	TUE	MAX165A			±1	LSB
		MAX165B			±2	
		MAX166A/C			±1	
		MAX166B/D			±2	

Figure 21-11: A few of the seemingly endless electrical characteristics.

Figure 21-12 and Figure 21-13 present the information we'll need to know in order to interface the RAT to this device. Figure 21-12 shows some timing information that is referenced from Figure 21-3. We're to the point now where we need to know how the ADC operates. The timing diagram of Figure 21-13 best describes the operation of the ADC, which is true of most timing diagrams with their weight in spit.

There are four signals of importance to this discussion: CS, RD, BUSY, and DATA. The first three signals are scalar (not buses) while the DATA signal is an 8-bit bus. The operation of the ADC is relatively simple. The ultimate goal is to obtain a digital number from the DATA lines. To do this, we need to start a conversion and wait for the information on the DATA outputs to be valid. The particular characteristics of the device are a function of the ADC and you must read the datasheet in order to figure out how the thing works. Once you know how it works, you must design the resulting hardware and firmware to actually use the device. An overview of the operating characteristics of the ADC is presented after Figure 21-12 and Figure 21-3.

MAX165/MAX166

TIMING CHARACTERISTICS (Note 6)

(V_{DD} = +5V, REFIN = +1.23V, AGND = DGND = 0V, unless otherwise noted.)

PARAMETER	SYMBOL	CONDITIONS	T_A = +25°C ALL		T_A = T_{MIN} to T_{MAX} MAX16_C/E		MAX16_M		UNITS
			MIN	MAX	MIN	MAX	MIN	MAX	
$\overline{CS}$ to $\overline{RD}$ Setup Time	t_1		0		0		0		ns
$\overline{RD}$ to $\overline{BUSY}$ Propagation Time	t_2			100		100		120	ns
Data-Access Time after $\overline{RD}$	t_3	(Note 7)		100		100		120	ns
$\overline{RD}$ Pulse Width	t_4		100		100		120		ns
$\overline{CS}$ to $\overline{RD}$ Hold Time	t_5		0		0		0		ns
Data-Access Time after $\overline{BUSY}$	t_6	(Note 7)		80		80		100	ns
Data-Hold Time after $\overline{RD}$	t_7	(Note 8)	10	80	10	80	10	100	ns
$\overline{BUSY}$ to $\overline{CS}$ Delay	t_8		0		0		0		ns

Note 1: Offset Error is measured with respect to an ideal first code transition which occurs at 1/2LSB.
Note 2: REFOUT is not available for use in MAX166C/MAX166D. These parts must be used with an external reference.
Note 3: Guaranteed by design, not tested.
Note 4: Accuracy may degrade at conversion times other than those specified.
Note 5: Power-supply current is measured when MAX165/MAX166 are inactive, i.e.
for MAX165 $\overline{CS}$ = $\overline{RD}$ = $\overline{BUSY}$ = high;
for MAX166 $\overline{CS}$ = $\overline{RD}$ = $\overline{BUSY}$ = MODE = high.
Note 6: Timing Specifications are sample tested at +25°C to ensure compliance. All input control signals are specified with t_r = t_f = 20ns (10% to 90% of +5V) and timed from a 1.6V voltage level.
Note 7: t_3 and t_6 are measured with the load circuits of Figure 1 and defined as the time required for an output to cross 0.8V or 2.4V.
Note 8: t_7 is defined as the time required for the data lines to change 0.5V when loaded with the circuits of Figure 2.
Specifications subject to change without notice.

Figure 21-12: Some of the timing characteristics of the ADC.

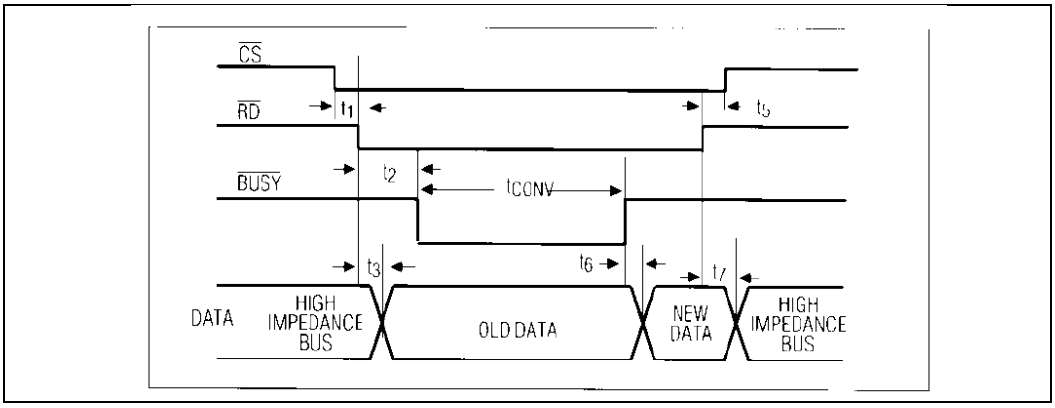


Figure 21-13: The all-important timing diagram associated with the ADC.

1. Start a conversion: The CS and RD signals are used to start a conversion. These two signals are active low. Examining the timing diagram in Figure 21-3 shows that dropping the two signals from their high state to their low state starts the conversion. Figure 21-3 indicates a slight delay between the dropping of the CS and RD signal (labeled t_1). Examination of the timing characteristics in Figure 21-12 shows that t_1 is actually 0ns so we don't need to worry about that.

2. Once the CS and RD signals become active (brought to their low state), the conversion has been requested. Sometime after the conversion request, the BUSY signal is driven low, which indicates that the conversion has officially started. The BUSY signal is an output of the ADC and is thus controlled by the inner-workings of the ADC. In other words, once the CS and RD signals have been driven low, the BUSY signal will automatically go low sometime later (t_2). Once again examining the timing characteristics shown in Figure 21-12, you can see that this time is 100ns.

3. Once the BUSY signal drops low, the analog-to-digital conversion is taking place inside of the ADC. The BUSY signal will stay low until the conversion is complete and will require no more than 5µs. When the conversion is complete, the BUSY signal goes high.

4. The data is not valid until shortly after the BUSY signal goes high. Figure 21-13 shows there is a slight delay, indicated by t_6, before the digital data is valid. Referring back to Figure 21-12, you can see that this value is 100ns. This once again is going to be an important value.

5. Once the conversion has completed, you can then read the DATA outputs of the ADC and be relatively sure that the data is valid.

6. The final step is to turn off the ADC. This is accomplished by bringing the CS and RD lines back to their high state. Note that when the CS and RD signals are not asserted, we consider the ADC to be turned off and the DATA outputs of the device go into a high-impedance state.

21.4.2 RAT Interface Specifics for the ADC

We're to the point where we'll assume we know how the ADC works and we're now ready to generate the actual hardware used to interface with the device. Figure 21-14 shows the signals on the ADC that we're particularly interested in for the interface requirements of the ADC. Note that many of the signals from the actual device are omitted for clarity but they were analog stuff anyway so they are inherently less interesting than the digital signals. Figure 21-15 shows the RAT block diagram once again. Both of these figures are important in the interfacing circuit of Figure 21-16.

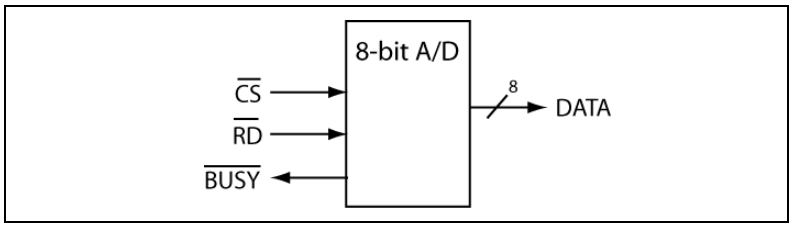


Figure 21-14: The signals of digital interest on the ADC.

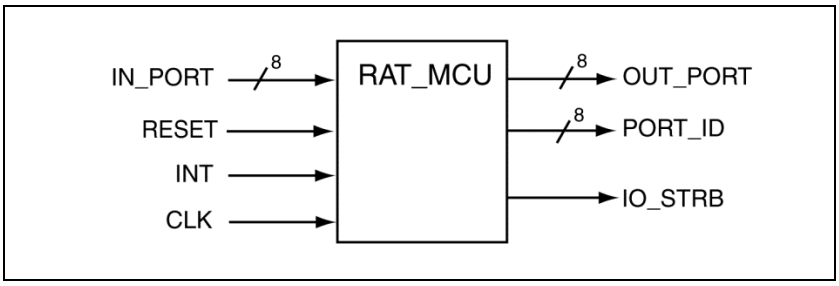


Figure 21-15: RAT black box diagram.

Figure 21-16 shows the circuit describing the critical interfacing for this problem. The WR_Strobe is the IO_STRB on the RAT MCU. Keep in mind that this is not the only way to interface with the ADC; there are many possible combinations. One more thing to keep in mind before we describe this interface is that the signal assignments on many parts of this circuit are arbitrary. Hopefully this circuit does not see too daunting; read the explanation following Figure 21-16 if it does.

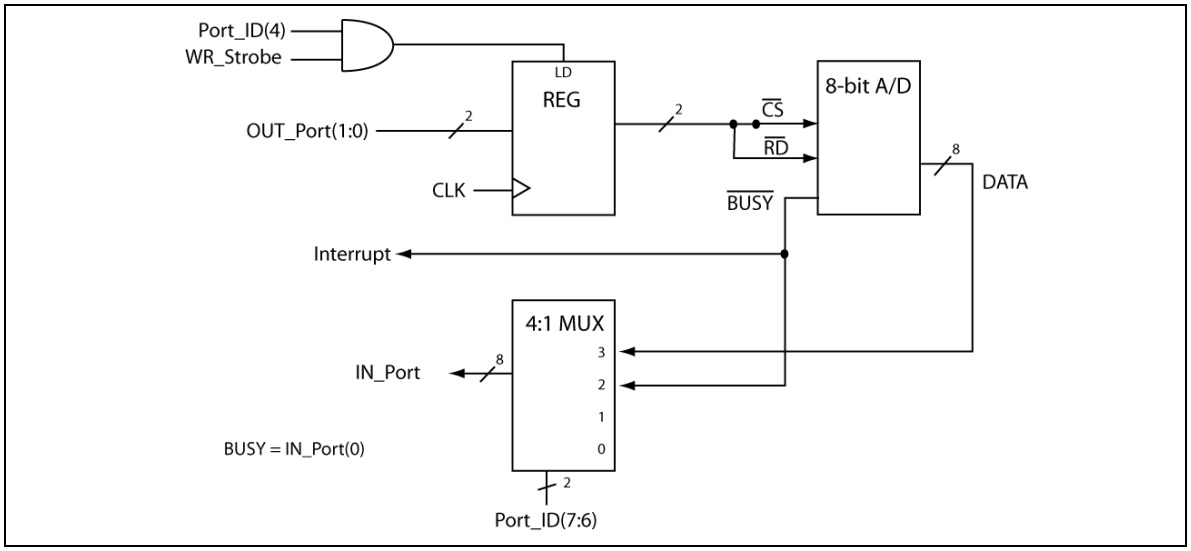


Figure 21-16: The final interface circuit including the ADC device.

- The CS and RD signals are registered. Loading of this register is controlled by the WR_Strobe and Port_ID(4) signal from the RAT. Keep in mind that the Port_ID(4) is one of the signals on the

Port_ID output of RAT as is shown Figure 21-16. Assignment of Port_ID(4) to control this register is arbitrary. The input of the register is OUT_Port signals from RAT.

- There are actually nine output signals from the ADC that we need to read: eight data outputs and the BUSY signal. This means that we'll need to use some type of data section device in order to read both of these values into RAT. What better device for data section than a MUX. A 4:1 MUX is used in Figure 21-16 with the control signals arbitrarily assigned to Port_ID(7:6).

- The BUSY signal is connected to the Interrupt input of RAT. This will allow the ADC to be interrupt driven which is a good thing for reasons stated later in a later problem. The BUSY signal is mapped to the LSB of the IN_Port signal. The output of the MUX is IN_Port signals of RAT.

- Table 21.2 shows the addresses used to access the hardware shown in Figure 21-16.

Constant Specifier	Value	Purpose
AD_CTRL	0x10	Used to write CS and RD values to ADC
AD_DATA	0xC0	Used to read data values from ADC
AD_BUSY	0x80	Used to read status of BUSY line of ADC

Table 21.2: Port addresses for the hardware specified in Figure 21-16.

21.4.3 ADC Interfacing: Writing the Firmware

Now that the hardware is setup, we're ready to write the firmware. The hardware was set up in order to be able to handle the I/O in two different modes: Polled and Interrupt driven. We'll be writing the firmware that describes both of these modes.

Example 21.2

Write a RAT subroutine that does an A/D conversion and stores the 8-bit result in register r10. Assume the RAT clock frequency is 50MHz.

The Polled Mode Solution: This is probably the most straightforward method to controlling the ADC. Remember that the overall goal here is to obtain a digital number from the ADC. The thing to remember about polling is that the microcontroller is dedicated to the task. In other words, it can do no useful work when the processor is polling. Figure 21-18 shows the code implementing a polled mode operation of the ADC.

As with all these types of problems, a good approach to generating a solution is to first generate a flow chart. For this problem, Figure 21-17 shows a flowchart for the polled mode solution. The code in Figure 21-18 works fine but… since the conversion requires 5μs and the RAT instructions require 40ns to execute, the program control will be stuck in the polling loop shown in Figure 21-18 for (5μs/40ns), or 125 instruction cycles. These instructions cycles effectively do nothing and could be used to do more useful processing. This highlights the disadvantage for polled I/O and the advantage of interrupt I/O as shown in the code of Figure 21-20. I hope that the listed comments tell the whole story. The flowchart provided in Figure 21-7 hopefully elicits some of the overall details.

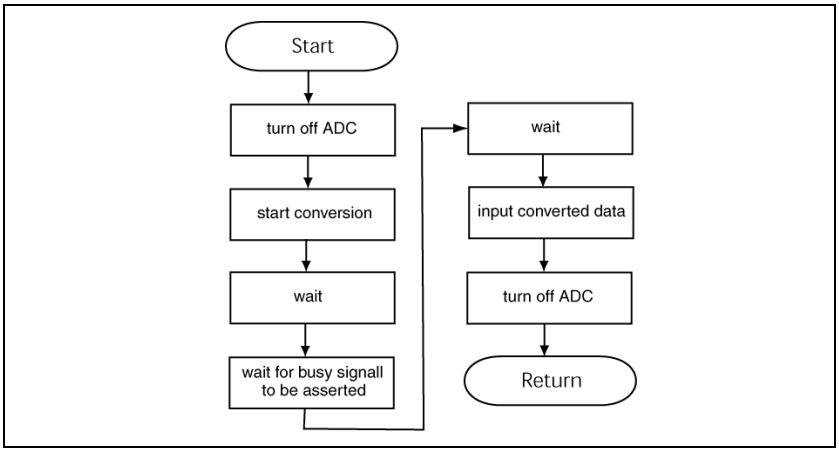


Figure 21-17: A flowchart for the polled mode solution.

```
;-----------------------------------------------------------------------
;- Assembler directives for control and data
;-----------------------------------------------------------------------
.EQU  AD_CTRL = 0x10     ; port address for A/D control signals (RD,CS)
.EQU  AD_DATA = 0xC0     ; port address for A/D DATA signals
.EQU  AD_BUSY = 0x80     ; port address for A/D status signal (BUSY)
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;- Subroutine: ad_conv_poll
;- This subroutine initiates a analog to digital conversion, waits
;- for the result (polled mode), and places the result in register r10
;-
;- Registers Used: r0
;-----------------------------------------------------------------------
ad_conv_poll:
          CLI                   ; make sure the interrupt is off since it
                                ;  is connected to the BUSY signal
          MOV      r0,0xFF      ; make sure ADC is off
          OUT      r0,AD_CTRL   ;
                                ;
          MOV      r0,0x00      ; turn on ADC and start conversion
          OUT      r0,AD_CTRL   ;
                                ;
          AND      r0,r0        ; delay (wait for BUSY signal to go low)
          AND      r0,r0        ;  these three instructions (at 40ns each)
          AND      r0,r0        ;  surpass 100ns delay requirement.
                                ;
poll:     IN       r0,AD_BUSY   ; Poll stuff: get BUSY signal
          TEST     r0,0x01      ; Mask bit0
          BREQ     poll         ; Keep polling until BUSY goes high
                                ;
          AND      r0,r0        ; Delay because data is not valid until 100ns
          AND      r0,r0        ;  after A/D conversion is complete (BUSY goes
          AND      r0,r0        ;  high.
                                ;
          IN       r10,AD_DATA  ; Get the conversion data from ADC.
                                ;
          MOV      r0,0xFF      ; Turn off the ADC.
          OUT      r0,AD_CTRL   ;
                                ;
          RET                   ; Pass control back to calling routine.
;-----------------------------------------------------------------------------
```

Figure 21-18: RAT assembly code that implements a polled mode control of the ADC.

Example 21.3

Write a RAT subroutine that does an A/D conversion and stores the 8-bit result in register r10. Assume the RAT clock frequency is 50MHz. Use an interrupt-driven approach for this solution.

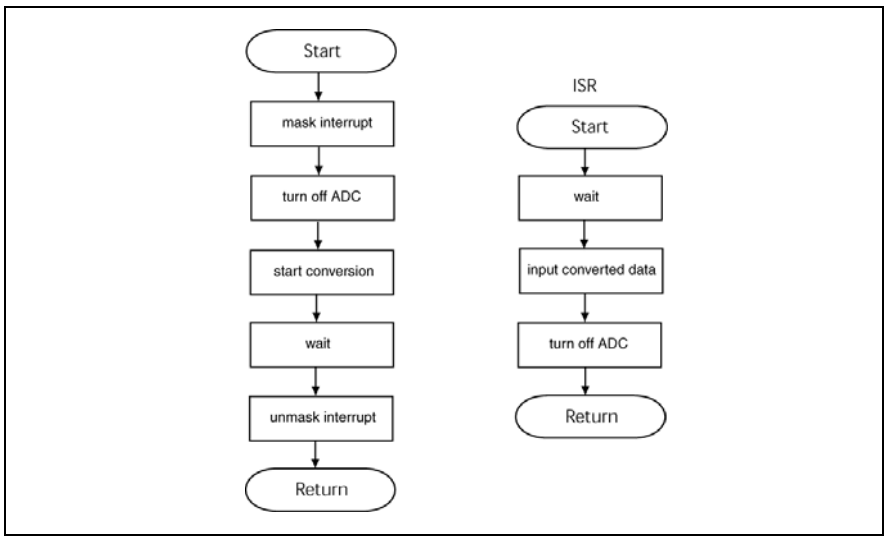


Figure 21-19: The flowchart for the interrupt-driven solution.

```
;-----------------------------------------------------------------------
;- Assembler directives for control and data
;-----------------------------------------------------------------------
.EQU  AD_CTRL = 0x10     ; port addr for A/D control signals (RD,CS)
.EQU  AD_DATA = 0xC0     ; port addr for A/D DATA signals
.EQU  AD_BUSY = 0x80     ; port addr for A/D status signal (BUSY)
;-----------------------------------------------------------------------

;-----------------------------------------------------------------------
;- Subroutine ad_conv
;- This subroutine initiates an analog to digital conversion, waits
;- for the result (polled mode), and places the result in register
;- r10.
;-----------------------------------------------------------------------
ad_conv_intr:
          CLI                   ; Make sure the interrupt is off
                                ;
          MOV     r0,0xFF       ; make sure ADC is off
          OUT     r0,AD_CTRL    ;
                                ;
          MOV     r0,0x00       ; turn on ADC and start conversion
          OUT     r0,AD_CTRL    ;
                                ;
          AND     r0,r0         ; delay (wait for BUSY signal to go low)
          AND     r0,r0         ;  these three instructions (at 40ns each)
          AND     r0,r0         ;  surpass 100ns delay requirement. Without
                                ;  delay, the interrupt would happen once it
                                ;  saw the BUSY signal in a high state
          SEI                   ; Allow for interrupts
          RET                   ; Pass control back to call routine.
;-------------------------------------------------------------------------------

;-------------------------------------------------------------------------
;- Subroutine: ISR
;- If this routine is called, the BUSY signal went high and the conversion
;- has completed. The converted data is stored in register r10.
;-------------------------------------------------------------------------
ISR:      CLI                   ; Make sure interrupt dead (a nop)
          AND     r0,r0         ; Delay (wait 100ns for the data to be valid
          AND     r0,r0         ;  after the BUSY signal goes high)
                                ;
          IN      r10,AD_DATA    ; Get the conversion data from ADC.
                                ;
          LOAD    r0,0xFF       ; Turn off the ADC.
          OUT     r0,AD_CTRL    ;
                                ;
          SEI                   ; Pass control back to instruction that was
                                ;  being executed when interrupt was received
;-------------------------------------------------------------------------------

.CSEG                            ; Make sure we're in the code segment
.ORG  0x3FF                      ; interrupt vector address
          BRN      ISR
```

Figure 21-20: RAT assembly code that implements an interrupt driven control of the ADC.

21.5 Chapter Summary

- Many microcontrollers, such as the RAT MCU are relatively simple devices. The MCUs have the ability to control things, but typically the things they control are not part of the MCU. Typical computer systems have added devices, which we typically refer to as external peripheral devices. MCUs are typically used to control these devices by synthesizing the required interface signals under program control.

- The steps required to interface MCU with external peripherals are:

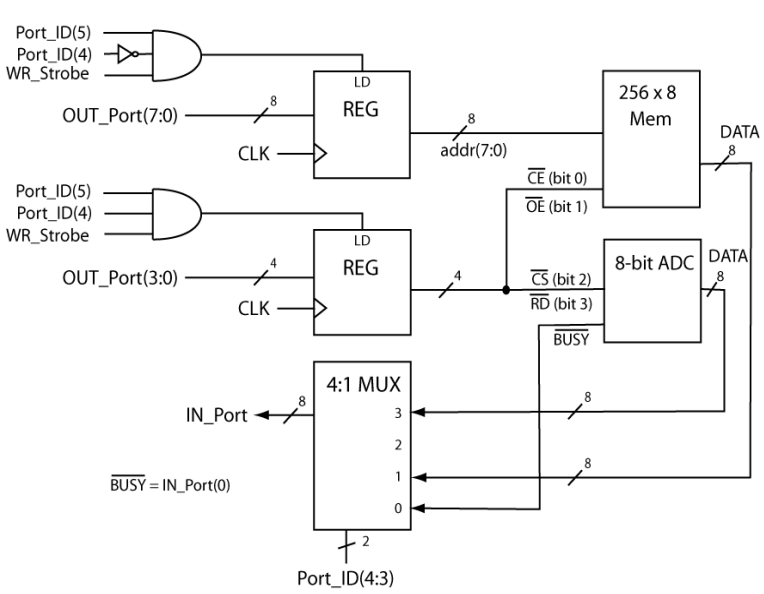
    1. Define device interface requirements
    2. Design required interface hardware
    3. Design required firmware to interface with hardware

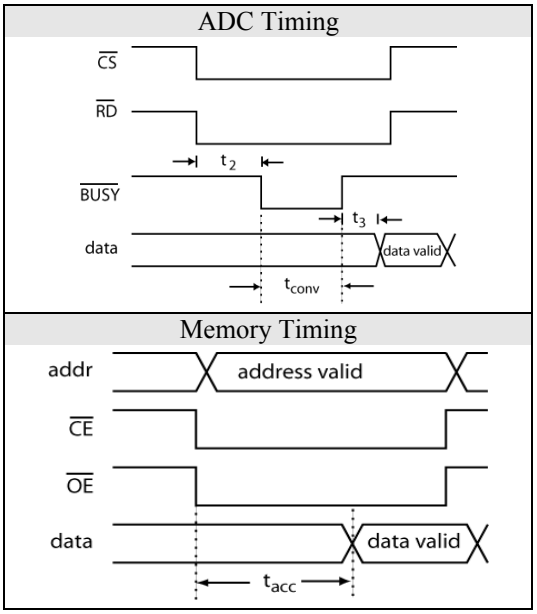
- Analog-to-digital converters provide common method for computer-type devices to interface to the analog world. These devices convert inherently analog values into their digital, or discrete, equivalents, which places I in a form that computers can work with.

21.6 Chapter Exercises

1) Write a RAT assembly language subroutine that does the following: Reads a value from the memory location stored in register r11. If the value read from memory is greater than 127_{10} than an analog to digital conversion is done and the result is stored in register r10. Otherwise, zero is stored in r10. *Minimize* the number of instructions used in your solution.

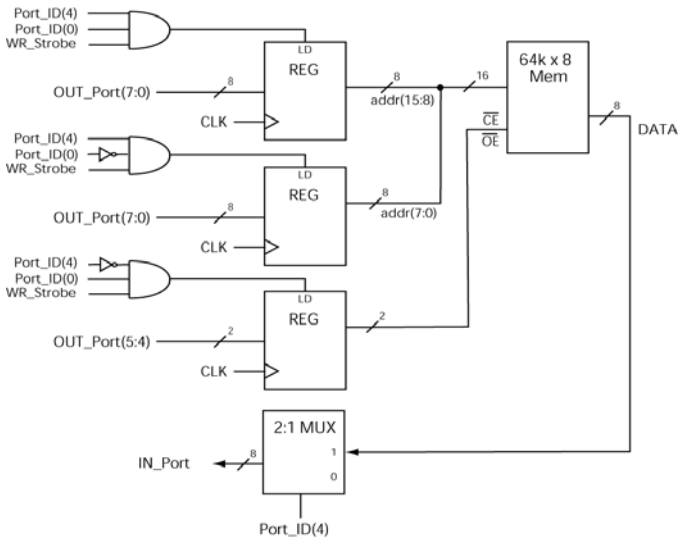
   - Assume a 100MHz RAT system clock (10ns period)
   - Assume $t_2 = t_3 = 35ns$; $t_{acc} = 110ns$; $t_{conv} = 10\mu s$ (where $ns = 10^{-9}s$ and $\mu s = 10^{-6}s$)
   - Assume the BUSY signal high-to-low transition is caused by dropping the CS and RD signals low; the BUSY signal low-to-high transition indicates the conversion has completed.



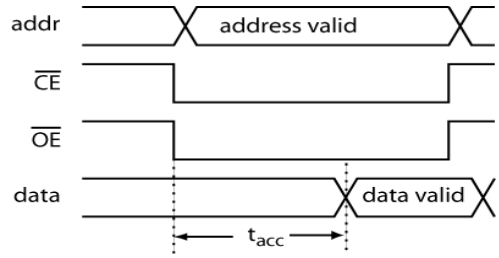


2) Write a RATMCU subroutine that reads from two *consecutive* memory locations, averages the data found in those memory locations, and places the result in register s8. The address of the first memory location is provided in register r10 (high address) and r11 (low address). Minimize the number of instructions you use in your code.

- Assume a 50MHz RAT MCU system clock (20ns period)
- Assume that the memory access time is 35ns (where ns = 10^{-9}s)
- Assume the high address will never equal 0xFF



Memory Timing

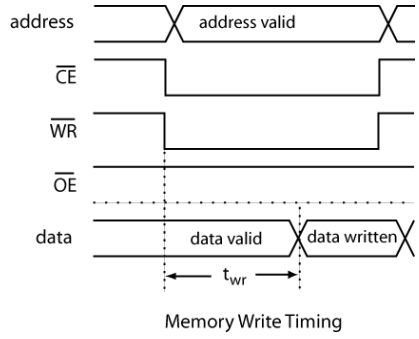



```
; fill these in

CONSTANT   mem_hi,
CONSTANT   mem_lo,
CONSTANT   mem_ctrl,
CONSTANT   mem_data,
```

3) RAMs are often used as *look-up-tables* or *LUTs*. In this way, a function's dependent values are stored in the LUT while the independent values are represented by the range of address locations of the RAM. For this problem, write a RAT subroutine that will write all the locations in the RAM with data generated by the function listed below. If the independent value overflows the 8-bit range, the memory location should be set to 0xFF. *Hint: start by writing a flowchart for a subroutine that exclusively writes a byte to memory.*

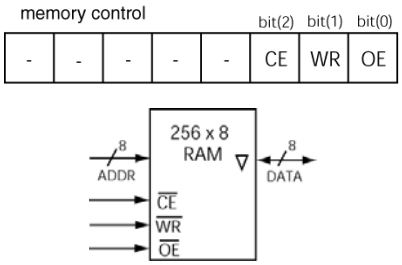
function: y = 1.5x + b;	b passed to subroutine in register r10 x is the independent variable (RAM address value

Assume: RAT clock frequency = 50MHz (20ns period); t_{wr} = 30ns



Memory Write Timing

memory address: Port 26
memory data: Port 36
memory control: Port 46




```
constant  mem_addr,  26
constant  mem_data,  36
constant  mem_ctrl,  46

constant  mem_write,
constant  mem_off,
```

22 VHDL TestBenches

22.1 Introduction

If we were all perfect, there would be no need for this chapter. However, since we all generally make mistakes, we definitely need a mechanism to find and correct those mistakes. Out there in digital design-land, the approach we take to creating digital circuits is 1) design them, 2) simulate them, and 3) implement them. The ordering of these steps is massively important despite the fact that most people do step 3) before doing step 2), if they even do step 2) at all.

You should simulate every circuit you design in order to increase your confidence level that the circuit is working properly. Life is easy with simple circuits; you can probably survive without testing them. More complex digital circuits inherently contain many nuances so that you simply can't assume they will work without somehow properly verifying that fact. This chapter presents the basic tools and theory of writing "testbenches", which is the term VHDL uses to describe a VHDL-based simulation mechanism for your circuit. This chapter is not an exhaustive approach to writing testbenches; it only aims to get you started. Once you get started, your circuit simulation techniques will quickly go far beyond the drivel in this chapter.

We opted to use a high chapter number for this chapter because there seemed to be no optimal location to present this material. If you don't want or need to test your VHDL models, you'll have no need for this chapter. However, any good digital designer knows that testing is an important part of designing digital circuits[1]. Additionally, you don't need to read this entire chapter; use what you need when you need it.

Main Chapter Topics

- **VHDL TESTBENCHES:** VHDL uses testbenches to verify circuit operation. This chapter presents an overview and introduction to VHDL testbenches.
- **TESTBENCH TEST VECTORS:** This chapter describes the options VHDL testbenches can use to generate and/or access data used by testbenches.
- **VHDL ASSERT STATEMENT:** This chapter describes how testbenches use assert statements to help verify proper circuit operation.
- **VHDL WAIT STATEMENTS:** This chapter describes the four types of VHDL wait statements and provides examples of their usage in actual testbench models.

Why This Chapter is Important

This chapter is important because it provides an overview and introduction to writing testbenches in VHDL. The VHDL language uses testbenches as a mechanism for verifying the proper operation of VHDL models using other VHDL models.

[1] If the circuits you design don't work, you may as well become an academic administrator instead. For such positions, no experience, expertise, nor intelligence of any kind is required.

22.2 Testbench Overview

Most of your VHDL career up to this point focused on designing circuits that would be synthesized, which is one of the powerful points of VHDL. However, keep in mind that one of the other powerful characteristics of VHDL is the ability to design models that can test other VHDL models. In other words, VHDL is such a versatile modeling tool that it can also act as a simulation mechanism. Unfortunately, in many instances, the main focus of digital design using VHDL is the design and generation of circuits; the testing/verification portion of circuit design is attenuated due primarily to time constraints.

Because testbenches are an important part of VHDL modeling, they are also an important part of the modern digital design process. In the initial stages of learning digital design, you're designs were most like simple enough so that you could verify their correct operation by examining the circuit models or testing the final circuit implemented on some type of real hardware. In all likelihood, you may not have even simulated your circuit. This is all fine, but the non-testing approach quickly breaks as your digital circuit become more complex.

As your digital designs become more complex, you're going to need to simulate them as part of the design process. In this context, simulation serves two purposes. First, simulation is going to be a great design tool. If you haven't realized it already, digital circuits can become complex. The complexity increases further when you are designing with non-ideal devices and you're forced to deal with the propagation delays associated with physical devices2. Secondly, simulation is a great debugging tool. If your circuit is not working and it's not obvious why, you'll know it's time to simulate. If you're truly doing the digital design thing correctly (and your designs are not trivial), you should be finding yourself spending as much time writing simulation models as you spend writing the hardware models themselves.

VHDL uses the term "testbench" to describe the mechanism VHDL uses to verify the functional correctness of your VHDL models. This chapter provides a vehicle to get you started writing testbenches, and thus allows you to verify the correct operation of your hardware models.

Finally, in the real world, the up-front verification of circuit operation is critical to the success of any project. As you know, the earlier you catch errors, the easier they are to fix and they'll have less tendency to generate more errors and induce bad design decisions along the way. This is massively important in the case of custom ASIC design when obtaining an actual design on silicon is going cost you about a million bucks3. In the end, if you play your cards right, you'll be using simulation as your primary design tool. Keep in mind that the original use of VHDL was as a tool to allow you to model and simulate digital designs.

22.3 Testbench Overview: VHDL's Approach to Circuit Simulation

A testbench can mean many different things. For this chapter, we'll consider a testbench to be a VHDL model that is separate from your VHDL circuit model. The testbench works in conjunction with the VHDL model with the purpose of verify proper operation of the VHDL model. The major difference between the testbench and the circuit you're testing is the fact that you probably intend to synthesize your circuit model while your testbench model is probably non-synthesizable. As you'll soon find out, the testbench models typically use VHDL constructs that don't synthesize. This is because the primary purpose of the testbench is to provide a set of "stimulus" to the model you're testing.

The testbenches we'll examine are written in VHDL. While this is not a requirement, there are many advantages to writing testbenches in the same language used to model the circuit. The main advantages are that you won't

2 The underlying thought here is that many digital circuits need to operate as fast as possible. In this case, many factors rear their ugly heads and conspire to undermine the proper operation of your circuit. All digital circuits stop functioning properly at some speed. Often times the goal in digital design is to push your circuit to operate as fast as possible. In many cases, making your circuit operate faster is a primary design constraint and will necessarily force you to redesign your circuit in order to meet required time constraints.

3 Which is why prototyping with PLDs is a powerful alternative to paying for a custom ASIC.

need to learn a new tool or a new language4, and the entity used to test your circuit can be included with and tested using the same tools you used to model your circuit.

Testbenches are a deep subject; there are many approaches to testbench design as result of flexibility of the VHDL language. This chapter will get you started on using testbenches to verify your design. As you continue your digital design journey, necessity will force you to learn many more circuit verification techniques not presented in this chapter.

22.4 The Basic Testbench Models

VHDL test benches can span the gamut from quite simple to massively complex depending on the intended purpose of the design. Often times, the testbench model can become more complicated than the actual circuit you're testing. The result is that there are many different "models" associated with testing a circuit with a testbench. This section provides a quick overview of some of the more popular models, which are the ones we'll crank though later in this chapter. This is the quick overview part, provided to give you a quick feel for testbenches. The low-level details arrive later.

Figure 22-1 shows the most basic testbench model. This model comprises of two main components: the "stimulus driver" and the "design under test" (DUT)5. These two boxes are typically referred to by many different names but the functions are still the same (so don't become too hung up on the names). The DUT is the VHDL model you're intending to test; the stimulus driver is a VHDL model that communicates with the DUT by providing it with inputs to exercise the DUT. Here are a few important things to note regarding Figure 22-1.

There is no magic in Figure 22-1. The truth is that the model depends on some human generating all the details of the "stimulus driver" box. You the human and you the designer of the circuit will need to decide what to test and determine if every important part of your circuit has been exercised by the stimulus driver enough for you to say, "yes, this circuit works"6.

The dotted box labeled "testbench" represents the testbench model in terms of the VHDL language. Note that the dotted box has no inputs or output; thus, the VHDL entity will have no inputs or outputs either. This is strange, but you'll quickly get used to it.

The testbench is really testing something. The model listed in Figure 22-1 will generally be used to generate a timing diagram. Using this model, the timing diagram will then need a visual inspection from some human in order to verify the circuit is working properly.

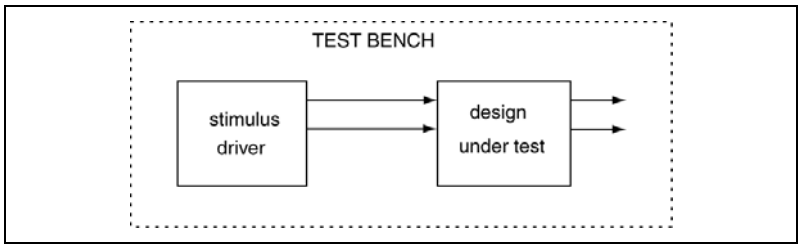


Figure 22-1: A block diagram for a basic VHDL testbench.

4 This is not completely true; you'll soon find out that we'll be using several features in VHDL that we have not previously used.
5 People sometimes refer to the DUT as a "UUT" which stands for "unit under test". Other times people refer to it as the "MUT", or model under test. Cool people refer to it as a "BBUT", or bowling ball under test. People sometime refer to the stimulus driver as the waveform generator.
6 Whereas an academic administrator would say, "yes, this is a circuit" as these people are deathly afraid of committing to anything (unless of course they can blame their failures on innocent people).

Figure 22-2 shows a slightly modified model of Figure 22-1. In many testbenches, the DUT sends feedback and/or intermediate results back to the stimulus driver. The stimulus driver can use this feedback as part of the testing (such as verifying the correctness of intermediate results) or sequencing of the testbench (waiting for a status signal indicating that some portion of the DUT is ready to be tested). The models of Figure 22-1 and Figure 22-2 are quite simple are relatively common.

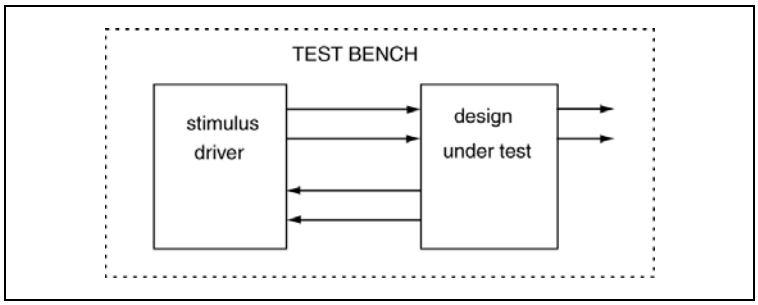


Figure 22-2: A block diagram for a more complicated VHDL testbench.

Figure 22-3 shows an extension of Figure 22-2. This figure shows that you can design your testbenches an extremely useful feature. As was present in Figure 22-2, the stimulus driver can contain pre-determined values that the model can use to verify the correctness of intermediate, or "expected" results. In this case, this set of predetermined results can determine whether the DUT passed the "test" or not. The notion here is that if all the results comparisons are happy, then the DUT passes the test. There are many methods you can use to implement the "results comparison" box of Figure 22-3.

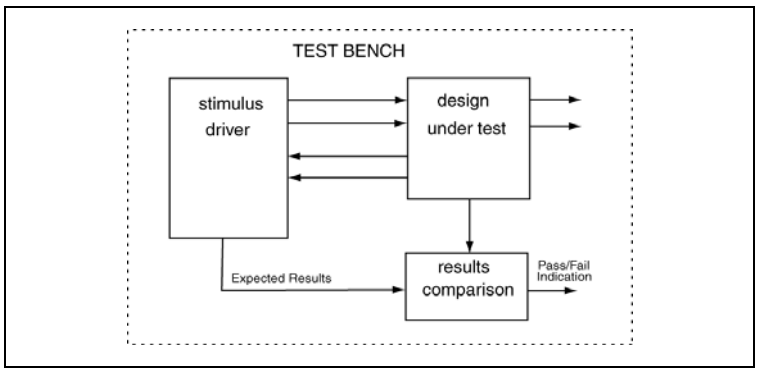


Figure 22-3: A block diagram for yet another VHDL testbench.

Figure 22-4 shows one final model we'll work with in this chapter. This model shows that the testbench can use external files for either storing the test vectors or writing the results of the test. Simple circuits don't require fancy testbenches, but as circuits become more complicated, other more clever approaches can be used to verify correct circuit operation.

One interesting thing to keep in mind here is that the external files for test vectors can be generated by some other test device such as the output of some other simulator you're using to test your design. Many times designs are first done on paper or modeled with computer programs; these devices can output test vectors more easily and completely than a human can do. VHDL has many facilities for accessing data from external files to use as test vectors7.

7 This chapter only describes a few of them.

As you know from previous models, the testbench can determine whether the DUT has passed the testing procedures or not. While a pass/fail indication is nice, it does not provided much information. When your circuit is large and complex, you'll surely want the testbench to give you information along the way, particularly if you design does not pass the testing procedures. Once again, VHDL has many options for outputting information to files. These files can then be compared directly to files generated by other test mechanisms to determine whether the DUT is working properly.

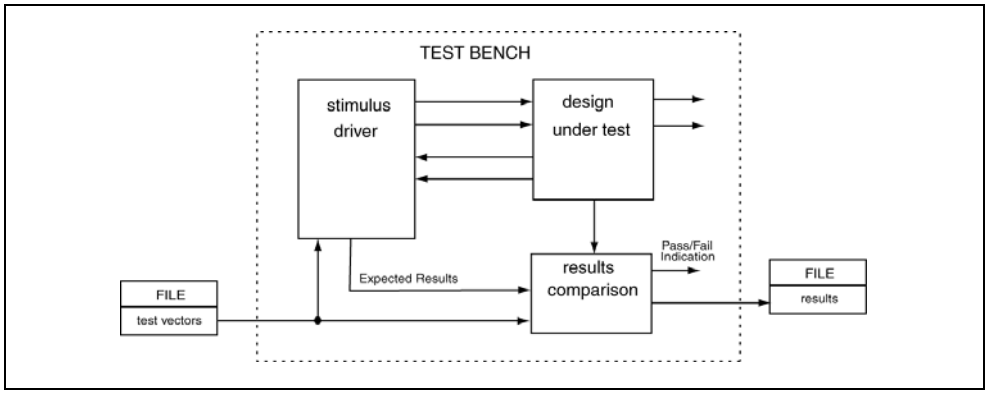


Figure 22-4: A block diagram for a VHDL testbench that reads test vectors from external files and writes the results to other external files.

VHDL has many facilities you can use to test your design. This chapter presents some of the basics regarding the use of testbenches for circuit verification. There is much more to the story, so if you find yourself being required to write amazing test benches, then you'll need to learn more or all of the details this chapter has opted to omit. VHDL is powerful, particularly in the context of circuit verification: learn what you need to as the need arises.

22.5 The Stimulus Driver

The heart of the testbench is the stimulus driver. This black box represents a majority of the code you'll be writing for your testbench. Being as important as it is, this section provides a short overview of possible structures for the stimulus driver. The basic structure of a testbench is simple but the issues are in the implementation details, which can seem daunting for anyone new to writing testbenches. Some of the details in this section were "hinted at" in the previous section in the provided black box testbench diagrams. This section paves the way for the examples we present later in this chapter.

You have two possibilities regarding the verification of a circuit. The stimulus driver of course generates the testing waveforms, but whose job is it to actually verify your circuit is working? Verification of the proper operation of a circuit can be either manual or automatic. The two main ideas here is whether you want a human to examine the resulting output waveforms from a testbench in order to verify proper operation (manual) or whether you want the testbench to automatically state whether the circuit is working properly. Here is an overview of what is good and bad about each approach.

- Manual Verification: This is the simplest approach and is thus the approach taken by most beginning testbench writers. The testbench models shown in Figure 22-1 and Figure 22-2 are examples of manual circuit verification. The models in these figures only generate the test vectors for the DUT, which inherently produce some output signals. The notion here is that the human reader will need to examine

the resulting waveforms in order to state whether the circuit is operating properly. This is a great option for beginners, but less great of an option if you have a complex circuit you need to test the crap out of[8].

- Automatic Verification: As your digital designs become more complex, you'll want to avoid manual verification. In truth, VHDL has many constructs that assist in the creation of testbenches that use automatic verification; this chapter presents only a few of them[9]. The testbench models in Figure 22-3 and Figure 22-4 show examples of automatic verification with the inclusion of the black box labeled "results comparison". There are many ways use VHDL code to structure the "results comparison" box; the examples in this chapter describe a few.

22.6 Vector Generation Possibilities

VHDL provides several approaches for your stimulus driver to supply test vectors to your DUT. More specifically, the list below describes the three main approaches to generating test vectors. Note that in any given testbench, you can you any combination of these three approaches.

Test Vectors generated "On The Fly": These vectors include any vectors that are not stored in internal VHDL structures (such as arrays) or stored in external files. This form of test vector generation works best for simple circuits or circuits that do not require extensive testing or complicated test vectors. In terms of simple stimulus drivers, you include whatever you need to in your VHDL code, but this approach quickly becomes unwieldy for large testbenches.

Test Vectors read from VHDL arrays: The VHDL language contains "arrays" which can be used to store constants. One approach to writing stimulus drivers is to store the test vectors as constants inside of arrays and access those constants using VHDL. The good part about this approach is that the test vectors will always be included with the testbench model (as opposed to reading test vectors from a file).

Test Vectors read from files: As with other computer languages, VHDL as the ability to read from and write to files. One of the main uses of this mechanism is storing testing information in external file. These external files include files where the test vectors will be read from and external files where the results will be written. The good part about this approach is that the test vectors can easily be generated from other computer programs such as a spreadsheet, some other type of simulator, or a higher-level language program you may be using to verify your circuit models.

22.7 Results Comparison: The "assert" Statement

The "assert" statement is both common and useful in the circuit verification using VHDL testbenches. The nice thing about assert statements is that they are simple to use and understand. There are many ways to verify the proper outputs from a DUT; using assert statements is one of those ways. An assert statement simply checks the Boolean value returned from the evaluation of the expression associated with the assert statement. If the expression evaluates as true, nothing happens. If the expression evaluates as false, the testbench provides information regarding of what went wrong.

Figure 22-5 shows an example of an assert statement. We'll see statements used in actual testbenches in a later section; this statement is shows the basic syntax of the statement. Here are a few important things to note about the assert statement shown in Figure 22-5.

8 Keep in mind that as digital circuits become more complex, you're going to need to write more complex testbenches in order to be 100% certain that they are working properly.
9 If you continue on in VHDL, you'll gather many more testbench writing skills. This chapter only aims to give you a quick taste of the possibilities.

The expression in the parenthesis returns a Boolean value. If this expression is true, nothing happens. If this expression evaluates as false, the items associated with the "report" and "severity" statements occur. The things that happen are the printing of the text associated with the "report" and "severity" lines to somewhere or something, which is typically the console window associated with the simulator exercising the testbench10.

The "report" and "severity" lines are associated with an assert statement. You can include one of both of these options in your testbench. The "severity" line output one of four piece of text: "Note", "Warning", "Error", or "Failure". The human writing the testbench chooses the appropriate text for the "severity" line. The "report" line prints out a user-specified message.

```
assert (s_signal = '1')
   report "s_signal does != 1; do something!
   severity Warning
```

Figure 22-5: A quick overview of the four main types of "wait" statements.

There is more to say about assert statements but we'll only mention one final notion. Although there is only one form of the assert statement, they are slightly different based on where in the testbench code they can appear. Assert statements can appear as concurrent statements or as sequential statements. If the assert statement appears as part of a process statement, it is a sequential statement; otherwise, it is a concurrent statement. The difference here is that assert statements as part of process execute when they are encountered as execution travels through the other sequential statements in the process. Concurrent assert statements execute anytime there is a change in the any of the signals present in the expression of the assert statement. In this way, the concurrent assert statement acts like a sensitivity list for a process statement. Though these difference seem subtle, they are actually quite significant.

22.8 The Process Statement: A Re-Visitation

Process statements: you've seen them before, but in only one form. Here's what you know about a process statement: when a signal in the sensitivity list of the process changes, the process "executes"11. When a process statement executes, it commences stepping through the sequential statement contained in the statement. When all of the statements in the process have completed execution, the process terminates. Another way of looking at this is that activity on a signal in the sensitivity list wakes up the process; the process then executes its statements until it reaches the ends of the process, then the process goes back to sleep (and waits for more action on a signal in the sensitivity list)12.

You've probably written about a bajillion process statements by this time in your digital design career. That truth is that all of the examples in this text used only one of two major forms of process statements. The form we've been using contains a sensitivity list to indicate which signals are important to the process. When the value of one of the signals on the sensitivity list changed, the process executes. This is the most common form of a process statement and is the form that you should use when you intend on synthesizing your circuit. The other form of a process statement does not contain a sensitivity list. As you may guess, this form is less useful than the other form for modeling circuits. This other process form, however, is quite useful when your VHDL code is modeling a testbench, so we'll describe it here in detail.

10 Depending on your particular simulator, other information is also include as part of assert statement failures.
11 I really don't like the word executes because it sounds way too much like you're describing the operation of a piece of computer code. VHDL, on the other hand, describes the operation of a piece of hardware.
12 This is knowingly a tough concept. I grapple with it constantly. I never quite understand it; I simply accept it. The notion of having sequential statements within a process statement and the fact that a process statement is a concurrent statement always thumps my brain. And in the end, you can use the model to generate hardware? This is sometime too much for my little brain.

As a quick example, Figure 22-6 shows a D flip-flop modeled both with and without a sensitivity list. The process in Figure 22-6(a) activates and executes when there is a change on either the CLK or D signal. The process complete execution, deactivates, and waits for more changes on CLK or D. The process in Figure 22-6(b) does not start execution until the wait statement is satisfied; when the rising edge of the clock occurs, the process activates, executes, suspends executions, and waits for the next rising clock edge. As you can see, it's not that big of a deal.

```
----------------------------------------
-- D flip-flop using a sensitivity list
----------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_nowait of DFF is
begin
   process (CLK,D)
   begin
      if (rising_edge(CLK)) then
         Q <= D;
      end if;
   end process;
end DFF_nowait;
```

(a)

```
----------------------------------------
-- D flip-flop using a sensitivity list
----------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait of DFF is
begin
   process
   begin
      wait until (rising_edge(CLK));
         Q <= D;
   end process;
end DFF_wait;
```

(b)

Figure 22-6: An example of a D flip-flop described without (a) and with (b) a wait statement.

The two forms of process statements are quite distinct. If your process statement does not include a sensitivity list, then it must include at least one wait statement13. If your process statement does include a sensitivity list, then the body of your process cannot include a wait statement. The sensitivity list of a process controls when the process will activate. In particular, when one of the signals on the sensitivity list changes, the process activates and the entire process executes from beginning to end. The notion of "from beginning to end" means that there is nothing to stop the process along the way. This is great for modeling circuits, but not so great for testing VHDL models.

On the other hand, process statements without sensitivity lists rely on "wait statements" to control the activation and deactivation of a process. Process statements using wait statements start and stop under control of the wait statements. The effect of this is that the process is either not executing (because it has been deactivated because of a wait statement") or the process is executing (because a wait statement condition was met and the process is "not waiting").

22.9 Attack of the Killer Wait Statements

This section covers the four forms of wait statements. Testbenches certainly don't always use all of these forms, but we include them here for both completeness and just in case you feel you need to use one of these forms. The following subsections provide a basic description and example of wait statements. Keep in mind that the basic function of a wait statement is to give you the ability to suspend execution of a process at any point in the process. This differs from statements using sensitivity lists in that once they start executing, they cannot be suspended until the end of the process is reached.

The four forms of a wait statement are straightforward in that the process is always "waiting" for something. Figure 22-7 shows the four things that can be "waited for"; the following sections expand on these basic wait statement types.

13 There are several forms of wait statements; we'll get to those soon.

a change in a signal (WAIT ON)

an expression is true (WAIT UNTIL)

a specific amount of time (WAIT FOR)

an eternity (WAIT)

Figure 22-7: A quick overview of the four main types of "wait" statements.

22.9.1 The "wait on" Statement

The "wait on" statement is the most similar wait statement to a process sensitivity list. The "wait on" statement is waiting for a change in the list of signal in the wait on statement. When one of the signals in the wait statement list changes, the process resumes executing with the statement following the "wait on" statement.

Figure 22-8 shows an example of D flip-flop model using a "wait on" statement. One thing to keep in mind is that process execution is circular in nature, meaning that when the process reaches the end, it automatically continues execution at the beginning of the process. The model in Figure 22-8 "waits" for changes in either the CLK or the D signal; when such a change occurs, the process restarts execution at the if statement.

Figure 22-8 is for example purposes only. It is not even close to being a good way to model a D flip-flop. We'll see better uses for "wait on" statements when we start back looking at testbenches.

```
-------------------------------------------------------
-- D flip-flop using a "wait on" statement
-------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait_on of DFF is
begin
   process
   begin

      if (rising_edge(CLK)) then
         Q <= D;
      end if;

      wait on CLK,D;

   end process;
end DFF_wait_on;
```

Figure 22-8: An example of a D flip-flop modeled using a "wait on" statement.

22.9.2 The "wait until" Statement

The "wait until" form of a wait statement instructs a process to suspend execution until the evaluation of the expression associated with the "wait until" statement returns a value of true. The catch here is that the "wait

until" statement requires that the associated expression return a Boolean value in order for the statement to be valid.

Figure 22-9 shows an example of D flip-flop model using a "wait until" statement. This process chooses to expand the "rising_edge" function to show that the "wait until" statement really does evaluate an expression. The line in the process body of shown in Figure 22-9 that is commented out is also a valid statement as the "rising_edge" function returns a Boolean value. The model in Figure 22-9 "waits" for two conditions to simultaneously occur: a change the clock signal (the CLK'EVENT[14]) and the current value of the CLK signal to be a '1'.

Once again, the D flip-flop model in Figure 22-9 is for example purposes only as it is not a good way to model a D flip-flop. We'll see better uses for "wait until" statements later in this chapter.

```
-------------------------------------------------------
-- D flip-flop using a "wait until" statement
-------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
               Q : out std_logic);
end DFF;

architecture DFF_wait_until of DFF is
begin
   process
   begin

      --wait until (rising_edge(CLK));

      wait until (CLK = '1' and CLK'EVENT);
         Q <= D;

   end process;

end DFF_wait_until;
```

Figure 22-9: An example of a D flip-flop modeled using a "wait until" statement.

22.9.3 The "wait for" Statement

The notion of a "wait for" statement means the associated process is forced to suspend execution and "wait" for the amount of time given in the "wait statement" argument. Once the stated amount of time has passed, the process resumes execution. Figure 22-10 shows the syntax for a "wait for" statement as well as a few examples.

```
wait for time_expression


Examples:

    -- using a value
    wait for 25ns;

    -- using a constant
    wait for (CLOCK_PERIOD);

    -- using an expression
    wait for (CLOCK_PERIOD * 60);
```

Figure 22-10: The syntax and examples of a "wait for" statement.

14 We refer to this as a "tick event"; this text did not cover this aspect of VHDL.

The problem is that I don't have a great example for it. In lieu of a great example, we'll use a crappy example instead. Figure 22-11 shows the crappy example I speak of. The problem with this example is that the model will not synthesize as you expect most models to. The reason I included this example is to highlight the differences between models intended for synthesis and models intended for simulation (testbenches). The model in Figure 22-11 is trying to be a D flip-flop but will not synthesize due to the "wait for" statement.

```
-------------------------------------------------------
-- D flip-flop using a "wait for" statement
-- WARNING: this model does not synthesize
-------------------------------------------------------
entity DFF is
   port ( CLK,D : in std_logic;
                Q : out std_logic);
end DFF;

architecture DFF_wait_for of DFF is
begin
   process
   begin

      -- this does not work!
      wait for 10ns;

      if (rising_edge(CLK)) then
         Q <= D;
      end if;

   end process;
end DFF_wait_for;
```

Figure 22-11: An example of a D flip-flop model attempting to use a "wait for" statement.

22.9.4 The "wait" Statement

The "wait" statement is the simplest of the wait-flavored VHDL statements. Testbenches use this statement in order to end a process. Consequently, the "wait" statement (with no arguments) is used to end the simulation process. Once again, we'll see examples of this when we start introducing some actual testbenches. Figure 22-12 shows the syntax of a "wait" statement; we include no examples.

```
wait;
```

Figure 22-12: The syntax for a "wait" statement.

22.10 Getting Your Feet Wet: Some Example Testbenches

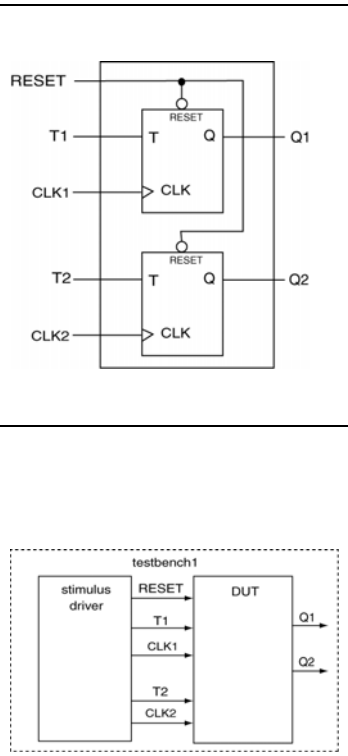
This section details some of the finer aspects of writing testbenches. This section by no means provides every possible testbench, but there should be enough information to get you started.

Example 22.1

Write a testbench that models the circuit shown in the top portion of Figure 22-13(b). Figure 22-13(a) provides the VHDL model for this circuit.

```
entity tff_ckt is
   port (   CLK1,CLK2 : in std_logic;
          RESET,T1,T2 : in std_logic;
                Q1,Q2 : out std_logic);
end tff_ckt;

architecture my_ckt of tff_ckt is
   signal s_tff_Q1, s_tff_Q2 : std_logic; --:='0';
begin

   tff1 : process (RESET,CLK1,T1)
   begin
      if (RESET = '0') then
         s_tff_Q1 <= '0';
      else
          if (rising_edge(CLK1)) then
              s_tff_Q1 <= T1 XOR s_tff_Q1;
          end if;
      end if;
   end process;

   tff2 : process (RESET,CLK2,T2)
   begin
      if (RESET = '0') then
         s_tff_Q2 <= '0';
      else
         if (rising_edge(CLK2)) then
            s_tff_Q2 <= T2 XOR s_tff_Q2;
         end if;
      end if;
   end process;

   Q1 <= s_tff_Q1;
   Q2 <= s_tff_Q2;

end my_ckt;
```

(a)



(b)

Figure 22-13: A pointless example circuit using T flip-flops; model (a) and block diagram (b).

Solution: Figure 22-14 shows the solution to Example 22.2

.

```
-------------------------------------------------------(1)
entity testbench1 is
end testbench1;

architecture stimulus of testbench1 is

   -- Misc declarations----------------------------------(2)
   constant CLK1_PERIOD: time := 40 ns;
   constant CLK2_PERIOD: time := 60 ns;

   -- declare DUT ---------------------------------------(3)
   component tff_ckt
      port (   CLK1,CLK2 : in std_logic;
             RESET,T1,T2 : in std_logic;
                   Q1,Q2 : out std_logic);
   end component;

   -- instantiate the device under test (DUT) ------------(4)
   signal s_t1, s_t2, s_q1, s_q2 : std_logic := '0';
   signal s_clk1, s_clk2 : std_logic := '0';
   signal s_reset : std_logic := '0';

begin
   -- instantiate the device under test (DUT) ------------(5)
   DUT: tff_ckt
   port map (CLK1 => s_clk1,
             CLK2 => s_clk2,
             RESET => s_reset,
             T1 => s_t1,
             T2 => s_t2,
             Q1 => s_q1,
             Q2 => s_q2);

   -- synthesize reset signal ----------------------------(6)
   s_reset <= '1', '0' after 20ns, '1' after 40 ns;

   -- set signal values ----------------------------------(7)
   s_t1 <= '1';    s_t2 <= '1';

   -- clk1 synthesis -------------------------------------(8)
   s_clk1 <= not s_clk1 after CLK1_PERIOD/2;

   -- clk2 synthesis -------------------------------------(9)
   clk2 : process
   begin
      wait for (CLK2_PERIOD * 0.75);
         s_clk2 <= '1';
      wait for (CLK2_PERIOD * 0.25);
         s_clk2 <= '0';
   end process;

end stimulus;
```

Figure 22-14: A block diagram for a basic VHDL testbench.

This is an instructive testbench model so we've included some fairly instructive comments. The following numbered items correspond to the numbers associated with the comment in Figure 22-14. Later testbench models will surely include less commentary (but probably not).

- This is the entity declaration; as advertised, there is no associated port clause as the testbench has no inputs and no outputs.

- This testbench uses some constants. As with programming using higher-level languages, the liberal use of constructs such as constant make your code more useful. This is particularly true when working with testbenches. If something changes, you don't want to search through a large VHDL model to make all the required modifications.

- This is the component declaration for the circuit that you intend on testing. Refer to the block diagrams in Figure 22-13(b) and realize that you've done this many times; it is no different when you're working with testbenches.

- This is a list of intermediate values used by the testbench. Note that these signals represent all of the signals shown in Figure 22-13(b). The important thing to notice here is that we assign initial values to all of the signals. This practice is almost a requirement, as you'll find out when you simulate circuits. If you do not assign these values, the simulator will not be smart enough to assign the values for you and you'll end up with unknown waveforms in your simulation results. If you assign these initial values later in your testbench model, these initialization values are overwritten.

- This is the instantiation of the device you're testing. Once again, this is nothing new and relation to the block diagrams in Figure 22-13(b).

- This is a concurrent statement used for a reset signal. This is a one-time statement so this implementation is probably the easiest approach. Note the statement is a list with commas (a VHDL feature we have not discussed). This statement uses the VHDL keyword "after" to provide the timing of this negative logic reset pulse. Note that the pulse is 20ns wide. This line executes three times during simulation: once to set the value, once to clear the value, and one more time to set the value again, which remains set throughout the simulation.

- This line contains two statements (in order to save space). These lines are concurrent statements that put the two outputs from the stimulus driver at known values. This line executes one time during the simulation.

- This is one approach to synthesizing a clock signal. This concurrent statement synthesizes a periodic clock signal (50% duty cycle) by toggling the signal every half CLK1 period. This concurrent statement is evaluated every half-CLK1 period throughout the simulation.

- This is another approach to synthesizing clock signals. This approach uses a process statement (without a sensitivity list) to generate the clock signal. This particular clock signal uses some special notation in order to provide CLK2 with a 25% duty cycle.

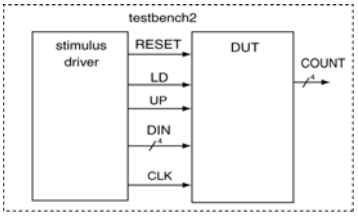
- For a final comment, the output of this testbench is the Q1 and Q2 signals. Because of the associated T flip-flops are always asserted, they act to halve the frequency of the two clock signals. For a given simulation, you'll probably include all the available signals in your waveform output.

Example 22.2

Write the VHDL code that implements the testbench model shown in the black box diagram of Figure 22-15(b). For this circuit, consider the DUT to be a 4-bit up/down counter with a synchronous parallel load (as modeled by Figure 22-15(a)).

```
entity COUNT_4B is
   port ( RESET,CLK : in std_logic;
               LD,UP : in std_logic;
                 DIN : in std_logic_vector (3 downto
0);
               COUNT : out std_logic_vector (3 downto
0));
end COUNT_4B;

architecture my_count of COUNT_4B is
   signal  s_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
   begin
      if (RESET = '1') then
         s_cnt <= (others => '0');
      elsif (rising_edge(CLK)) then
         if (LD = '1') then     s_cnt <= DIN;
         else
            if (UP = '1') then  s_cnt <= s_cnt + 1;
            else                s_cnt <= s_cnt - 1;
            end if;
         end if;
      end if;
   end process;

   COUNT <= s_cnt;

end my_count;
```

(a)



(b)

Figure 22-15: An example using a 4-bit counter; the VHDL model (a) and a diagram showing the testbench circuit. (b).

Solution: This example problem really did not ask much. We are using this problem as a stepping-stone for presenting meaningful testbench modeling concepts. Figure 22-16 shows an example testbench for this problem. Here are a few meaningful comments corresponding to the numbered comments of Figure 22-16.

Although the intermediate signals were initialized when they were declared, this statement immediately changes them. The word “immediate” is important. In this context, it means the values change at time “zero” in the simulation. This is because the statements described by this comment are concurrent statements.

You saw this statement used previously as a reset signal. The important thing to notice here is that the width of the resulting pulse is 20ns (40ns – 20ns), and not 40ns as this line of code may lead you to think. This represents a use of “absolute time” in the testbench. Testbenches are typically easier to write and understand when they use “relative time”, which is why this chapter uses relative time so extensively.

This code synthesizes a load pulse for the counter. Once again, the code uses absolute time, so the width of this load pulse is 30ns.

```
entity testbench2 is
end testbench2;

architecture stimulus of testbench2 is

   constant CLK_PERIOD: time := 50 ns;

   component COUNT_4B
   port ( RESET,CLK,LD,UP : in std_logic;
                    DIN : in std_logic_vector (3 downto 0);
                  COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up, s_ld : std_logic := '0';
   signal s_clk      : std_logic := '0';
   signal s_reset    : std_logic := '0';
   signal s_ld_val   : std_logic_vector(3 downto 0) := X"0";
   signal s_count    : std_logic_vector(3 downto 0) :=  X"0";

begin
   -- instantiate the counter
   DUT: COUNT_4B
   port map (CLK => s_clk,
             DIN => s_ld_val,
             RESET => s_reset,
             LD => s_ld,
             UP => s_up,
             COUNT => s_count);

   -- clock synthesis
   s_clk <= not s_clk after CLK_PERIOD/2;

   ------------------------------------------------------------(1)
   -- set initial signal values
   s_up <= '1';      s_ld_val <= X"E";

   ------------------------------------------------------------(2)
   -- synthesize reset signal
   s_reset <= '0', '1' after 20 ns, '0' after 40 ns;

   ------------------------------------------------------------(3)
   -- synthesize ld signal at 50ns
   s_ld <= '0', '1' after 50 ns, '0' after 70 ns;

end stimulus;
```

Figure 22-16: A block diagram for a basic VHDL testbench.

Example 22.3

Write the VHDL code that implements the testbench model shown in the black box diagram of Figure 22-15(b). For this circuit, consider the DUT to be a 4-bit up/down counter with a synchronous parallel load (as modeled by Figure 22-15(a)).

Solution: There is an issue with the approach of the solution to Example 22.2
The issue is that beyond the setting initial values of signal and synthesizing one-time signals (such as resets) was that the testbench started becoming somewhat unintuitive. The primary reason for this issue was that we were not able to use “wait” statements in the solution because the solution did not use process statements. When we use process statements, basic testbench writing becomes more intuitive and thus easier to write.

Figure 22-17 shows another testbench that exercises the 4-bit up/down counter. This solution uses both behavioral and dataflow statements in the testbench model. The fact that opted to put a majority of the "test" portion of the testbench in a process statement allows us to use wait statements. This testbench uses a process statement to perform most of the work. Note that there are two concurrent statements in this testbench: one generates the clock and the other does the testing work. This advertises the notion that concurrency in VHDL modeling also extends to testbenches. Once you model the clock synthesis, you're free to do the other test; the clock remains running. There are some other items worth noting in this testbench; the numbers below match the comments in the testbench model.

- The testbench uses a process statement. Because we want to use wait statements in this solution, the process statement does not include a sensitivity list.

- The first real work we do in the process statement is to reset the counter by sending it an active high pulse. Keep in mind that the reset signal is initialized to its non-asserted state. The statements associated with this comment say that the reset signal toggles at the beginning of the simulation and then toggles again 20ns later, resulting in a 20ns pulse.

- This code synthesizes a signal for a parallel load pulse. The important thing to note here is that the model is now working with relative time due to the nature of the "wait" statements. According to the given code, the load pulse is a 25ns wide positive pulse that occurs starting at 70ns. The 70ns time comes from the fact that this set of code waits 50ns after the 20ns delays associated with the reset pulse.

- This is a "wait until" statement; the process will pause at this statement until the condition associated with the "wait until" statement occurs. In this case, the counter is counting up and the wait statement "waits" until the counter reaches 10. At that time, the UP signal toggles, which forces the counter to count down (starting at the next clock cycle). This is important in terms of testbenches because what the model is doing is "waiting" for something to occur; when it occurs, the testbench assigns a value to some other signal associated with the testbench.

- This is an example of an assert statement (though not a great example). If execution of the process arrives at the assert statement, then we know the counter has passed the previous "wait until" statement. In this case, we know the counter to be at 10. The assert statement is saying, "if the counter is not 11 at this point, then print out this rude message". Assert statements are massively useful; this example will not be winning any awards. Figure 22-18 shows the resultant error message. Note that the simulator that generated this output included extra information in addition to the error message and severity level.

- The process ends with a final "wait" statement. This statement assures that no more statements in this process will evaluate by forcing the process to "wait" forever at this statement.

```
entity testbench3 is
end testbench3;

architecture stimulus of testbench3 is

   constant CLK_PERIOD: time := 40 ns;

   component COUNT_4B
   port ( RESET,CLK,LD,UP : in std_logic;
                    DIN : in std_logic_vector (3 downto 0);
                  COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up, s_ld : std_logic := '0';
   signal s_clk : std_logic := '0';
   signal s_reset : std_logic := '0';
   signal s_ld_val : std_logic_vector(3 downto 0) := X"0";
   signal s_count : std_logic_vector(3 downto 0) :=  X"2";

begin
   DUT: COUNT_4B -- instantiate counter
   port map (CLK => s_clk,
             DIN => s_ld_val,
             RESET => s_reset,
             LD => s_ld,
             UP => s_up,
             COUNT => s_count);

   -- clk synthesis
   s_clk <= not s_clk after CLK_PERIOD/2;

   ----------------------------------------------------(1)
   process
   begin
      s_up <= '1';   s_ld_val <= X"2";

      ----------------------------------------------------(2)
      -- synthesize reset signal
      s_reset <= '1';
      wait for 20 ns;
      s_reset <= '0';

      ----------------------------------------------------(3)
      -- synthesize loading signal
      wait for 50 ns;
      s_ld <= '1';
      wait for 25 ns;
      s_ld <= '0';

      ----------------------------------------------------(4)
      wait until s_count = X"A";
      ----------------------------------------------------(5)
      assert (s_count = X"B")
         report "This is a stupid test"
         severity Error;
      s_up <= '0';

      ----------------------------------------------------(6)
      wait;
   end process;
end stimulus;
```

Figure 22-17: A block diagram for a basic VHDL testbench.

Lastly, for this example, Figure 22-18 shows a sample result of an example where the “assert” statements did not evaluate as being true. In this case, the testbench code shown in Figure 22-17 “waited” until the count output was a certain value and then tested to see if it was a different value. This assert statement fails and the error message

shown in Figure 22-18 is printed to the console15. Note that my particular simulator includes the exact time in the simulation where the error occurs.

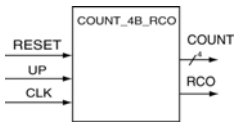
```
# ** Error: This is a stupid test
#    Time: 420 ns  Iteration: 2  Instance: /testbench3
```

Figure 22-18: A block diagram for a basic VHDL testbench.

Example 22.4

Write the VHDL code that implements a testbench model that verifies the up and down RCO output on the 4-bit counter model shown below.

```
entity COUNT_4B_RCO is
   port (RESET,CLK,UP : in std_logic;
                  RCO : out std_logic;
                COUNT : out std_logic_vector (3 downto
0));
end COUNT_4B_RCO;

architecture my_count of COUNT_4B_RCO is
   signal  t_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
   begin
       --RCO <= '0';
       if (RESET = '1') then
          t_cnt <= (others => '0');
       elsif (rising_edge(CLK)) then
          RCO <= '0';
          if (UP = '1') then
             t_cnt <= t_cnt + 1;
             if (t_cnt = X"E") then
                RCO <= '1';
             end if;
          else
             t_cnt <= t_cnt - 1;
             if (t_cnt = x"1") then
                RCO <= '1';
             end if;
          end if;
       end if;
   end process;

   COUNT <= t_cnt;

end my_count;
```



Solution: Figure 22-19 shows one possible solution this example. This solution is similar to the previous solution so we'll only comment on the new and significant items.

The testbench initial resets the counter and starts the counter counting in the "up" direction. At that point, the testbench waits for the RCO signal to assert. Since the counter is counting in the up direction, the RCO signal should reset when the counter reaches 0xF. The test bench uses the "wait until" statement to monitor the RCO statement. When the RCO signal asserts, the first thing we need to do is wait a small amount of time; which is done with the following "wait for" statement. We need to do this because the simulator interprets the activity

15 This message is a result from the simulator I'm using. Your results may be different depending on the simulator that you're using.

happening on the RCO and COUNT output signal to occur simultaneously. After the small delay, the testbench then verifies that the count does indeed match the max count value of 0xF.

What we want here is have the counter continue counting in the up direction for about half of the count cycle. After that point we'll change the direction of the count (COUNT = '0') and verify the RCO is working in the down counting direction. This piece of code is another approach to inserting a delay in the testbench. Though we could have used a "wait for" statement here, we opted to use a loop construct. As of this writing, this text has not mentioned loops in VHDL, but they exist. This is an example of a simple look construct; you should find these straightforward. They are handy when you're creating complicated testbenches.

This piece of code changes the direction of the counter to count down after counting in the up direction for a few clock cycles.

This code is similar to the code described in (1) above. This time, however, we are verifying that the counter displays a 0x0 when the RCO is asserted when the counter is counting in the down direction.

```
entity testbench4 is
end testbench4;

architecture stimulus of testbench4 is

   constant CLK_PERIOD: time := 40 ns;

   component COUNT_4B_RCO
   port ( RESET,CLK,UP : in std_logic;
                  RCO : out std_logic;
                COUNT : out std_logic_vector (3 downto 0));
   end component;

   signal s_up : std_logic := '1';
   signal s_clk : std_logic := '0';
   signal s_reset : std_logic := '0';
   signal s_rco : std_logic := '0';
   signal s_count : std_logic_vector(3 downto 0) :=  X"0";

begin
   DUT: COUNT_4B_RCO -- instantiate counter
   port map (CLK => s_clk,
             UP => s_up,
             RESET => s_reset,
             RCO => s_rco,
             COUNT => s_count);

   s_clk <= not s_clk after CLK_PERIOD/2;

   process
   begin

      -- synthesize reset signal
      s_reset <= '1';
      wait for 20 ns;
      s_reset <= '0';

      ----------------------------------------------------(1)
      wait until s_rco = '1';
      wait for 1 ns;
         assert (s_count = X"F")
         report "RCO counting up is incorrect"  severity Error;

      ----------------------------------------------------(2)
      for M in 1 to 10 loop
         wait until rising_edge(s_clk);
      end loop;

      ----------------------------------------------------(3)
      s_up <= '0';

      ----------------------------------------------------(4)
      wait until s_rco = '1';
      wait for 1 ns;
         assert (s_count = X"0")
         report "RCO counting down is incorrect"  severity Error;

      wait;
   end process;
end stimulus;
```

Figure 22-19: A block diagram for a basic VHDL testbench.

The following comments deal with a slightly advanced issue in VHDL. Some people may not have read about or dealt with the issue of “signals” vs. “variables”. If that is the case then you can skip the next few items and simply move onto the next examples. The notion of “signals” vs. “variables” is not overly complicated, so

another option for you is to go read about the difference between signals and variables. The following discussion briefly mentions these differences.

In truth, the counter model shown in Example 22.4
has some issues that we needed to deal with in the testbench model. The problem with the counter model is that the original model was somewhat confusing because it was implemented using signals. Figure 22-20 shows a clearer version of the counter. This difference between this model and the model provided as part of Example 22.4
is that the model uses a variable for the temporary count value. Because the results from operations done on variables are ready immediately, the model can look for the true RCO values for both the up and down counts. Note that in the model of Example 22.4
, we had to generate the RCO signal one count in advance; in particular, “0xE” or “0x1” for the up and down counting directions, respectively. The two models are functionally equivalent, but the counter model shown in Figure 22-20 provides much less confusion.

```
entity COUNT_4B_RCO_VAR is
   port ( RESET,CLK,UP : in std_logic;
                  RCO : out std_logic;
                COUNT : out std_logic_vector (3 downto
0));
end COUNT_4B_RCO_VAR;

architecture my_count of COUNT_4B_RCO_VAR is
   signal  t_cnt : std_logic_vector(3 downto 0);
begin
   process (CLK, RESET)
      variable v_cnt : std_logic_vector(3 downto 0);
   begin
      if (RESET = '1') then
         v_cnt := (others => '0');
      elsif (rising_edge(CLK)) then
         RCO <= '0';
         if (UP = '1') then
            v_cnt := v_cnt + 1;
            if (v_cnt = X"F") then
               RCO <= '1';
            end if;
         else
            v_cnt := v_cnt - 1;
            if (v_cnt = x"0") then
               RCO <= '1';
            end if;
         end if;
      end if;
      t_cnt <= v_cnt;
   end process;

   COUNT <= t_cnt;
end my_count;
```

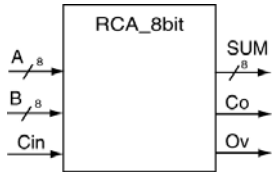
Figure 22-20: A clearer version of the RCO counter of the previous example.

Figure 22-21 shows a model for an 8-bit ripple carry adder. We’ll use this model for the next few example testbenches. One thing worth noting in this example is that the RCA contains an overflow output in addition the carry-out output. For this RCA model, we’ve defined a carry out to be the condition where the sign bit of the two operands are equivalent but differs from the sign-bit of the result.

The next few testbenches test the RCA using the same set of test vectors. As you’ll see, the main differences between the upcoming examples is where they obtain the test vectors from. There are several other differences between the upcoming examples; the example solutions briefly describe these differences.

```
entity RCA_8bit is
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
end RCA_8bit;

architecture RCA_8bit of RCA_8bit is
begin

   process(A,B,Cin)
      variable v_res : std_logic_vector(8 downto 0);
   begin
      v_res := ('0' & A) + ('0' & B) + Cin;
      Ov <= '0';

      if (A(7) = B(7)) then
         if (A(7) /= v_res(7)) then
            Ov <= '1';
         end if;
      end if;

      SUM <= v_res(7 downto 0);
      Co <= v_res(8);
   end process;

end RCA_8bit;
```

(a)



(b)

Figure 22-21: The RCA model used as the DUT for the next three examples.

Example 22.5

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 22-21. The testbench should use "on the fly" test vectors; use three sets of test vectors to test the RCA.

Solution: The approach taken by this problem is to generate the test vectors "on the fly". This is definitely the most straightforward approach, but it is not well suited for all testbenches. In particular, generating test vectors "on the fly" is only feasible for testbenches that are relatively short and don't have the need to exercise a large set of test vectors.

Figure 22-21 shows the final testbench for this example problem. There are a few relatively interesting things to note about this solution:

The approach here is to provide the values directly to the augend and addend. These values are essentially "hardcoded" into the VHDL model. These hardcoded values create the notion of the test vectors are "on the fly". After the assignment of the operand values, the testbench "waits" using a wait statement. After the expiration of the wait statement, the testbench verifies that the sum, the overflow, and the carry-out values are correct. The verification of these values is also hardcoded and thus modeled as "on the fly".

This represents the testing of the second set of test vectors. This is similar to the previous comments above, so not much to say here.

This represents the testing of the third and final set of test vectors. Again, not much to say here.

```
entity testbench6 is
end testbench6;

architecture stimulus of testbench6 is

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
              Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin : std_logic := '0';
   signal s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_Ov,
             Co => s_Co,
             SUM => s_sum);

   process
   begin

      s_cin <= '0';
      wait for 20 ns;

      ------------------------------------------------------------(1)
      s_A <= X"C3";      s_B <= X"54";
      wait for 50 ns;
      assert (s_sum = X"17")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '0')
         report "overflow is not correct"  severity Error;
      assert (s_co = '1')
         report "carry-out is not correct"  severity Error;

      ------------------------------------------------------------(2)
      s_A <= X"82";     s_B <= X"84";
      wait for 50 ns;
      assert (s_sum = X"06")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '1')
         report "overflow is not correct"  severity Error;
      assert (s_co = '1')
         report "carry-out is not correct"  severity Error;

      ------------------------------------------------------------(3)
      s_A <= X"85";     s_B <= X"24";
      wait for 50 ns;
      assert (s_sum = X"A9")
         report "SUM is not correct"  severity Error;
      assert (s_ov = '0')
         report "overflow is not correct"  severity Error;
      assert (s_co = '0')
         report "carry-out is not correct"  severity Error;

      wait;
   end process;
end stimulus;
```

Figure 22-22: A solution for Example 22.5

The testbench model shown in Figure 22-22 is an implementation of the testbench model shown in Figure 22-3 and repeated in Figure 22-23 for your viewing convenience. The solution to Example 22.5 effectively uses assert statements in order to implement the “results comparison” shown in Figure 22-23. In the context of Figure 22-23, the solution to Example 22.5

does not actually have an official “pass/fail indication”. What will occur is that the assert statements will print a message to the console if there is an error.

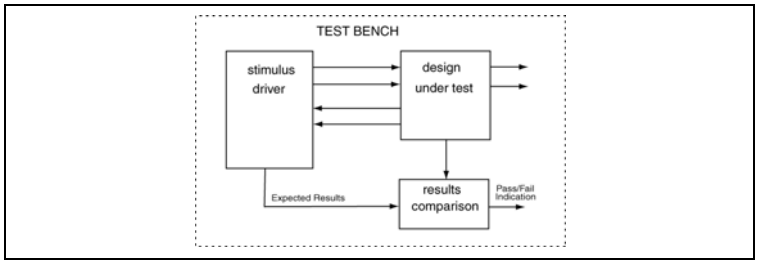


Figure 22-23: A block diagram for a basic VHDL testbench.

Example 22.6

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 22-21. The testbench should use test vectors that are stored in an array object; use three sets of test vectors to test the RCA. The testbench should verify that the value of the SUM output of the RCA is correct and base the success of the testing on only the SUM value.

Solution: This example problem once again verifies the proper operation of the RCA. This example uses test vectors stored in arrays included in the testbench model as opposed to the “on the fly” test vector storage of the previous example. You should note that though these test vectors are stored in arrays, they’re still “hardcoded” into arrays. The main advantage this approach has over the true “on the fly” approach is that all the test data is in one location instead of spread out through your code as it was in the “on the fly” approach. Figure 22-25 shows a solution to this example; here are a few more items to note in this solution.

VHDL allows the use of arrays, but it considers arrays to be a “type” of their own. This means that you need to explicitly describe arrays before you use them. This line of code defines arrays of size “three” in both vector and scalar forms; the vector arrays hold sum and operand inputs while the scalar arrays hold values for the carry-out and overflow. The use of the constant for the array size represents generic programming practices and is therefore happy.

This set of code defines constants of the array types defined in the previous step. Note that we’ve used hex notation for the 8-bit vectors to help the code appear neater. Also, note that these five array declarations use the two array definitions defined in the previous step.

This testbench uses a signal to state whether the DUT is working properly or not. Somewhat peculiar is the fact that this testbench only tests the results of the addition operation but does not check the carry-out and overflow results. Thus, the final verification is based on whether the SUM is correct but not the carry-out and overflow16. Note that we initialized the s_fail signal to ‘0’.

In order to simplify this testbench, we placed a bulk of the testing inside of a VHDL iterative construct. This testbench uses a VHDL looping construct, which is good approach based on the notion that we need to perform the same testing but on different sets of vectors. Note that looping construct iterates from ‘0’ to ‘2’, which highlights the notion that arrays in VHDL are zero-based.

16 This was an arbitrary choice, and not a good one at that.

This code tests whether the results of the addition operation are correct or not. If the SUM is not correct in any set of test vectors, the s_fail signal is asserted (s_fail = '1'). One way a user will be able to discern whether the test passed or failed is to monitor the value of the s_fail signal; the other way is to monitor the results of the assert statements.

This set of code is similar to the previous examples in that the testbench uses assert statements to indicate if there is an error in some RCA operation. If there is an error, the assert statements print a message out to a console window in the test environment.

Finally, the solution to this example once again uses the model shown yet again in Figure 22-24. The only comment here is that the official "pass/fail" indication will be a signal on a waveform display in the simulator you're using for your testing. Nothing too fancy here.

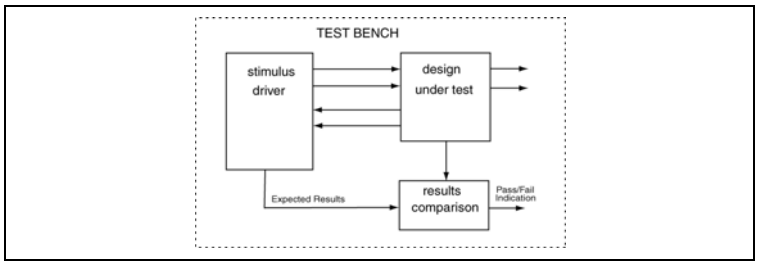


Figure 22-24: A block diagram for a basic VHDL testbench with "result comparison" capabilities.

```
entity testbench7 is
end testbench7;

architecture stimulus of testbench7 is
   constant VEC_NUM : integer := 3;

   ------------------------------------------------------------(1)
   type vec_arr is array (0 to (VEC_NUM-1)) of std_logic_vector(7 downto 0);
   type bit_arr is array (0 to (VEC_NUM-1)) of std_logic;

   ------------------------------------------------------------(2)
   constant A_arr: vec_arr := (X"C3", X"82", X"85");
   constant B_arr: vec_arr := (X"54", X"84", X"24");
   constant Sum_arr: vec_arr := (X"17", X"06", X"A9");
   constant co_arr: bit_arr := ('1', '1', '0');
   constant ov_arr: bit_arr := ('0', '1', '0');

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
              Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

   ------------------------------------------------------------(3)
   signal s_fail : std_logic := '0';

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_Ov,
             Co => s_Co,
             SUM => s_sum);

   process
   begin
      s_cin <= '0';
      wait for 20 ns;

      ------------------------------------------------------(4)
      for N in 0 to (VEC_NUM-1) loop

         s_A <= A_arr(N);     s_B <= B_arr(N);
         wait for 50 ns;

         ------------------------------------------------------(5)
         if (s_sum /= Sum_arr(N)) then
            s_fail <= '1';
         end if;

         ------------------------------------------------------(6)
         assert (s_sum = Sum_arr(N))
            report "SUM is not correct"  severity Error;
         assert (s_ov = ov_arr(N))
            report "overflow is not correct"  severity Error;
         assert (s_co = co_arr(N))
            report "carry-out is not correct"  severity Error;

      end loop;

      wait;
   end process;
end stimulus;
```

Figure 22-25: The solution to Example 22.6

.

Example 22.7: Test Vectors Stored in External Files

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 22-21. The testbench should use test vectors that are stored in an external file; use three sets of test vectors to test the RCA. The testbench should verify that the value of the SUM output of the RCA is correct and base the success of the testing on only the SUM value.

Solution: This example is similar to the previous example but the test vectors are stored in a file rather than "on the fly" or in arrays. Figure 22-26 shows a printout of the test vector file accessed by this example. There are a few things to note about this file:

The file uses a '$' delimiter to indicate that the given line is a comment and thus should be ignored by the testbench. As you'll see from the testbench model, this choice of comment symbols is arbitrary.

Each non-comment line in the test file contains five data fields: three 8-bit values and two 1-bit values. We arbitrarily chose to delineate the individual data fields using a single space. We could have opted to more space between data fields or none at all; the testbench code can handle either case based on the mechanism the testbench model uses to access the test vector file.

```
$-----------------------------------------------
$ tb8.txt: Sample test vector file
$
$ file format (name(width):
$
$     A(8), B(8), Sum(8), Co(1), Ov(1)
$
$-----------------------------------------------
11000011 01010100 00010111 1 0
10000010 10000100 00000110 1 1
10000101 00100100 10101001 0 0
```

Figure 22-26: A file of test vectors for use by Example 22.7.

There are many new and happy things to note about the solution Example 22.7 shown in Figure 22-28. Here are a few of those things as referenced in the testbench code.

- This testbench reads the input vectors from a file, so we first must open the associated file. This statement declared a "file" object as a text type, associated the type with the file named "tb8.txt", and opens that file in a "read" mode. The label "f_test_vects" is essentially a file handle that the testbench uses to retrieve information from the file.

- In most cases regarding testbenches, variables are more intuitive to work with. The issue is that we need the associated values immediately; we don't want to insert wait statements in order to force an update of values represented by signals. You'll assuredly be working more with variables in your testbenches as opposed to signals. Get used to it.

- The testbench reads data from the file one line at a time. The testbench code reads a line from the test vector file and places it into a variable of a "line" type, which is named "v_vec_line". Note the use of the file handle in this line.

- It's always good practice to comment things, even files containing test vectors. VHDL has no concept of how a test vector file will indicate a comment line. What this line does is essentially treat every line in the test vector file as a comment line if the first character in the line is a '$'. The choice of '$' is completely arbitrary. Note the VHDL keyword "next" executes if the testbench finds a comment character.

- This set of lines reads the test vectors from the "line" that was previously read from the test vector file. This data is read directly into the associated variables, which essentially has the effect of assigning the values to those variables. The associated hardware then acts on this new information and generates results.

Figure 22-28 shows the final result to Example 22.7. The solution uses a small font size and does not use comments in an effort to keep the solution less than one page. Figure 22-27 shows the high-level block diagram model of the testbench (we originally presented this diagram earlier in this chapter). As with the previous example, the official "pass/fail" indication will be a signal on a waveform display in the simulator you're using for your testing.

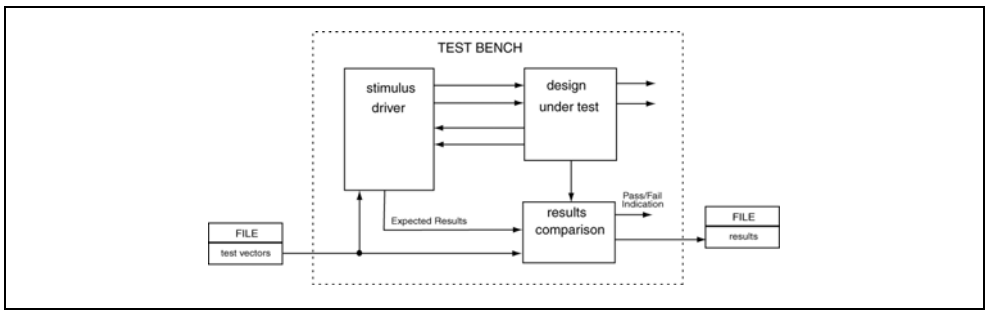


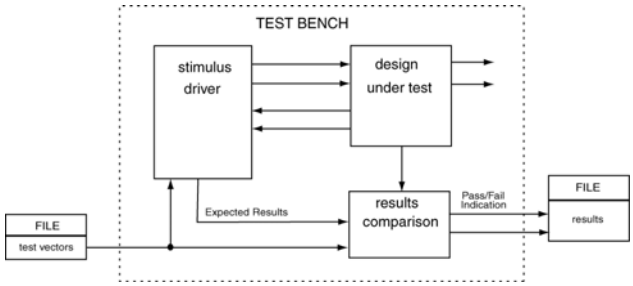
Figure 22-27: A block diagram for testbench solution of Example 22.7.

```
entity testbench8 is
end testbench8;

architecture stimulus of testbench8 is
   ------------------------------------------------------(1)
   file f_test_vecs: text open read_mode is "tb8.txt";

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
             Ov : out std_logic;
             Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

   signal s_fail : std_logic := '0';

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (A => s_A,
             B => s_B,
             Cin => s_cin,
             Ov => s_ov,
             Co => s_co,
             SUM => s_sum);

   process
      ----------------------------------------------------(2)
      variable v_vec_line: line;
      variable v_A, v_B : std_logic_vector(7 downto 0);
      variable v_sum_test : std_logic_vector(7 downto 0);
      variable v_ov_test, v_co_test : std_logic;
   begin

      s_cin <= '0';     wait for 20 ns;

      while not endfile(f_test_vecs) loop

         ------------------------------------------------(3)
         readline(f_test_vecs, v_vec_line);

         ------------------------------------------------(4)
         if v_vec_line(1) = '$' then
            next;
         end if;

         ------------------------------------------------(5)
         read(v_vec_line,v_A);        read(v_vec_line,v_B);
         read(v_vec_line,v_sum_test);
         read(v_vec_line,v_co_test);
         read(v_vec_line,v_ov_test);

         s_A <= v_A;     s_B <= v_B;
         wait for 50 ns;

         if (s_sum /= v_sum_test) then
            s_fail <= '1';
         end if;

         assert (s_sum = v_sum_test)
            report "SUM is not correct"  severity Error;
         assert (s_ov = v_ov_test)
            report "overflow is not correct"  severity Error;
         assert (s_co = v_co_test)
            report "carry-out is not correct"  severity Error;

      end loop;
      wait;
   end process;
end stimulus;
```

Figure 22-28: The solution to Example 22.7.

Example 22.8: Vectors Stored in External Files

Write the VHDL code that implements a testbench that can verify operation of the 8-bit RCA modeled Figure 22-21. Model your testbench solution using the following testbench model. Use the file shown in Figure 22-26 for the test vectors. Make sure verification of the DUT will fail if any of the DUT's outputs are not correct.



Solution: This example is similar to the previous example but there are a few extra items. First, from examining the testbench diagram shown in the problem description, you can see that this testbench will be different from the testbench of the previous example because the pass/fail indication must be written to a file. The problem statement also states that we need to verify every operation associated with the RCA; recall from the previous example that we only verified the results of the addition operation.

Figure 22-29 shows a solution to Example 22.8:. As always, there are many items worth noting about the solution shown in Figure 22-28. Here are a few of those things as referenced using the parenthetical comments in the testbench code.

This testbench reads the input vectors from a file and writes the results to another file. The testbench opens both of these files as text files, the test vector file in read mode, and the result file in write mode.

This set of code replaces the assert statements that essentially performed the same function in the solution to the previous example. This testbench writes error messages to the previously opened results file; recall that the assert statements write messages to the console. This set of code include the time with the use of the VHDL "now" keyword. Note that the sum, carry-out, and overflow results as verified with this set of code. Also, note that if any of these results are not correct, the entire test fails as noted by the assignment of the "v_fail" signal in each of the "if" cases in this set of code. The last thing to note regarding this set of code is that the solution writes to a previously declared "line" type, and then write that entire line to the output file. Note that successive writes to a "line" type are appended to previous writes to that line. The "writeline" statement writes the line to the output file, which additionally has the affect of clearing the "line" type buffer. In the way, the next "write" statement writes to an empty "line" buffer.

This set of code verifies checks the "v_fail" variable to discern whether the testbench was able to verify proper operation of the DUT. Note that anytime there was an error detected, the "v_fail" signal was set to true.

```
entity testbench9 is
end testbench9;

architecture stimulus of testbench9 is
   ------------------------------------------------- ----------------(1)
   file f_test_vecs: text open read_mode is "tb8.txt";
   file f_test_results: text open write_mode is "tb9_results.txt";

   component RCA_8bit
   port (   A,B : in std_logic_vector(7 downto 0);
            Cin : in std_logic;
          Ov,Co : out std_logic;
            SUM : out std_logic_vector(7 downto 0));
   end component;

   signal s_A, s_B : std_logic_vector(7 downto 0) := (others => '0');
   signal s_cin, s_ov, s_co : std_logic := '0';
   signal s_sum : std_logic_vector(7 downto 0) := (others => '0');

begin
   DUT: RCA_8bit -- instantiate the RCA
   port map (s_A, s_B, s_cin, s_ov, s_co, s_sum);

   process
      variable v_fail : std_logic:='0';
      variable v_vec_line: line;
      variable v_results_line: line;
      variable v_A, v_B : std_logic_vector(7 downto 0);
      variable v_sum_test : std_logic_vector(7 downto 0);
      variable v_ov_test, v_co_test : std_logic;
   begin

      s_cin <= '0';
    wait for 20 ns;

      while not endfile(f_test_vecs) loop

         readline(f_test_vecs, v_vec_line);

         if v_vec_line(1) = '$' then
            next;
         end if;

         read(v_vec_line,v_A);         read(v_vec_line,v_B);
         read(v_vec_line,v_sum_test);
         read(v_vec_line,v_co_test);
         read(v_vec_line,v_ov_test);

         s_A <= v_A;     s_B <= v_B;
         wait for 50 ns;

         ---------------------------------- ---- ------------- -----(2)
         if (s_sum /= v_sum_test) then
            write(v_results_line,String'("SUM is not correct at "));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;
         if (s_ov /= v_ov_test) then
            write(v_results_line,String'("Overflow is not correct "));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;
         if (s_co /= v_co_test) then
            write(v_results_line,String'("Carry_out is not correct"));
            write(v_results_line,now);
            writeline(f_test_results,v_results_line);
            v_fail := '1';
         end if;

      end loop;

    ---------------------------------------------  -- - ------------(3)
    if (v_fail = '1') then
       write(v_results_line,String'("The DUT failed testing."));
       writeline(f_test_results,v_results_line);
    else
       write(v_results_line,String'("The DUT tested properly."));
       writeline(f_test_results,v_results_line);
    end if;

      wait;
   end process;
end stimulus;
```

Figure 22-29: A solution to Example 22.8:.

22.11 Chapter Summary

- A “testbench” is the term VHDL uses to name a VHDL model whose job it is to verify the proper operation of a given digital circuits. The term “device under test”, or DUT, is used to refer to the circuit being tested by a given testbench. The form of a testbench is such that the DUT is instantiated and thus becomes part of the testbench. VHDL testbenches are comprised of VHDL code but unlike the DUT, the testbench is not synthesizable. The two main components of a testbench are the DUT and the stimulus driver.

- VHDL testbenches verify proper DUT operation by either manual or automatic testing. The person writing the testbench decides on an appropriate type of testing. Manual testing requires a human to examine the waveforms generated by the testbench to verify proper circuit operation. Automatic testing uses the versatility of VHDL to have the testbench model state directly whether the DUT model operates as expected.

- Test vectors for VHDL testbenches come from one of three sources: 1) “on the fly”, 2) from composite VHDL structures (such as arrays), or 3) from external files. Each of these sources are considered to emanate from the stimulus driver box of the testbench.

- VHDL can use “assert” statement to verify proper circuit operation. The assert statement consists of “report” lines and/or “severity” lines. If the expression associated with the assert statement does not evaluate as true, the VDL code executes the “report” and “severity” lines.

- VHDL process statements have two forms: either they use a sensitivity list or they use wait statements, but they cannot use both.

- There are four different types of wait statements: 1) wait for a change in a signal (WAIT ON), 2) wait until an expression evaluates are true (WAIT UNTIL), 3) wait for a specific amount of time (WAIT FOR), and 4) wait forever (WAIT).

22.12 Chapter Exercises

1) What are the two main purposes for simulating your digital circuits?

2) When will you know it's time to break down and simulate your digital circuit?

3) Briefly describe the two main approaches your testbench can use in order to verify a circuit is operating as expected.

4) Briefly describe the three main approaches to generating test vectors in a VHDL testbench.

5) Briefly describe the primary difference between an "assert" statement and an "if" statement?

6) List all the types of VHDL provided error messages that are associated with "severity" lines in VHDL "assert" statements.

7) Briefly describe whether is it possible to use neither a "report" line nor a "severity" line when using "assert" statements.

8) Are VHDL "assert" statements considered sequential statements or concurrent statements? Briefly describe the skinny on this notion.

9) Briefly describe the main differences between the two forms of wait statements.

10) Can a process statement include both a sensitivity list and a wait statement? Briefly describe why or why not.

11) How many types of wait statements are included in VHDL and what are they?

12) Consider a process statement that uses a "wait" statement. Briefly describe what happens when execution of the sequential statements in the process reaches the end?

13) Write a testbench that would completely test the following VHDL model. Use “on the fly” test vectors and use manual verification in your solution.

```
-----------------------------------------------------------------
-- RET D Flip-flop model with active-low synchronous set input.
-----------------------------------------------------------------
entity d_ff_ns is
    port (D,S,CLK : in  std_logic;
                Q : out std_logic);
end d_ff_ns;

architecture my_d_ff_ns of d_ff_ns is
begin
   dff:  process (D,S,CLK)
   begin
      if (rising_edge(CLK)) then
         if (S = '0') then
            Q <= '1';
         else
            Q <= D;
         end if;
      end if;
   end process dff;
end my_d_ff_ns;
```

14) Write a testbench that would completely test the following VHDL model. Use test vectors from arrays and use manual verification in your solution.

```
-----------------------------------------------------------------
-- RET T Flip-flop model with active-low asynchronous set input.
-----------------------------------------------------------------
entity t_ff_s is
    port ( T,S,CLK : in  std_logic;
                Q : out std_logic);
end t_ff_s;


architecture my_t_ff_s of t_ff_s is
   signal s_tmp : std_logic; -- intermediate signal declaration
begin
   tff:  process (T,S,CLK)
   begin
      if (S = '0') then
         Q <= '1';
      elsif (rising_edge(CLK)) then
         s_tmp <= T XOR s_tmp; -- temp output assignment
      end if;
   end process tff;

   Q <= s_tmp; -- final output assignment

end my_t_ff_s;
```

15) Write a testbench that would completely test the following VHDL model. Use test vectors from arrays and use automatic verification in your solution.

```
-------------------------------------------------------------
-- Model for a universal shift register
-------------------------------------------------------------
entity univ_sr is
    port (    SEL : in std_logic_vector(1 downto 0);
           P_LOAD : in std_logic_vector(7 downto 0);
            D_OUT : out std_logic_vector(7 downto 0);
              CLK : in std_logic;
            DR_IN : in std_logic;  -- input for shift left
            DL_IN : in std_logic); -- input for shift right
end univ_sr;

architecture my_sr of univ_sr is
   signal s_D : std_logic_vector(7 downto 0);
begin

   process (CLK,SEL,DR_IN,DL_IN,P_LOAD)
   begin
      if (rising_edge(CLK)) then

         case SEL is
            -- do nothing (don't change state) -------------
            when "00" =>  s_D <= s_D;

            -- parallel load --------------------------------
            when "01" =>  s_D <= P_LOAD;

            -- shift right ----------------------------------
            when "10" =>  s_D <= DL_IN & s_D(7 downto 1);

            -- shift left -----------------------------------
            when "11" =>  s_D <= s_D(6 downto 0) & DR_IN;

            -- default case ---------------------------------
            when others => s_D <= "00000000";
         end case;

      end if;
   end process;

   D_OUT <= s_D;

end my_sr;
```

16) Write a testbench that would completely test the following VHDL model. Use test vectors from external files and use automatic verification in your solution.

```
entity my_fsm2 is
    port (   TOG_EN : in  std_logic;
            CLK,CLR : in  std_logic;
               Y,Z1 : out std_logic);
end my_fsm2;

architecture fsm2 of my_fsm2 is
   type state_type is (ST0,ST1);
   signal PS,NS : state_type;
begin
   sync_proc: process(CLK,NS,CLR)
   begin
     if (CLR = '1') then
        PS <= ST0;
     elsif (rising_edge(CLK)) then
        PS <= NS;
     end if;
   end process sync_proc;

   comb_proc: process(PS,TOG_EN)
   begin
      case PS is
      Z1 <= '0';

         when ST0 =>    -- items regarding state ST0
            Z1 <= '0';  -- Moore output
            if (TOG_EN = '1') then NS <= ST1;
            else  NS <= ST0;
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (TOG_EN = '1') then NS <= ST0;
            else  NS <= ST1;
            end if;
         when others => -- the catch-all condition
            Z1 <= '0';  -- arbitrary; it should never
            NS <= ST0;  --  make it to these two statement
      end case;
   end process comb_proc;

   -- assign values representing the state variables
   with PS select
      Y <= '0' when ST0,
           '1' when ST1,
           '0' when others;

end fsm2;
```

17) Write a testbench that would completely test the following VHDL model. Use test vectors from external files and use automatic verification in your solution.

```
entity clk_div is
    Port (    clk : in std_logic;
           div_en : in std_logic;
            sclk : out std_logic);
end clk_div;


architecture my_clk_div of clk_div is

   type my_count is range 0 to 100;        -- user-defined type
   constant max_count : my_count := 63;    -- user-defined constant

   signal tmp_sclk : std_logic;            -- intermediate signal for clock

begin
   my_div: process (clk, div_en)

      variable div_count : my_count := 0;

   begin
      if (rising_edge(clk)) then    -- look for clock edge
         if (div_en = '1') then     -- divider enabled
            if (div_count = max_count) then
               tmp_sclk <= not tmp_sclk;    -- toggle output
               div_count := 0;      -- reset count
            else
               div_count := div_count + 1;
            end if;
         else                       -- divider disabled
            div_count := 0;         -- reset count
            tmp_sclk <= '0';        -- turn off output
         end if;
      end if;
   end process my_div;

   s_clk <= tmp_sclk;   -- assign to output

end my_clk_div;
```

Appendix

Requiem for the Digital Logic Designer

Digital design is the process where you create a digital circuit to solve a given problem. There are many approaches you can use to solve given problems, designing a digital logic circuit is simply one of them. What makes digital design so useful is that the design can generally interface with other digital circuits such as computer-type circuits. The two basic tenets of digital logic are:

- **Digital logic circuits are hierarchical**: We can describe a digital circuit at various levels; the level at which we describe digital logic is generally the one that allows us to transfer as much useful information as possible. Abstracting digital designs to higher levels aids in understanding and designing circuits.

- **Digital logic circuits are decomposable into a few basic digital circuits**: Although there are many ways to describe digital circuits, we strive to make the descriptions an aggregate compilation of standard digital circuits in able to help us understand the circuits.

A given digital design solves problems by having the outputs react to the inputs in a manner such that it solves the given problem. There are two basic types of digital logic circuits:

- **Combinatorial Circuits**: circuit outputs are a function of the circuit's inputs.

- **Sequential Circuits:** circuit outputs are a function of the sequence of the circuit's inputs.

The main ramification of sequential circuits is that they can "remember" the previous "state" of the circuit. Sequential circuits can store (remember) bits; we refer to the bits the circuit is remembering as the "state" of the circuit. Combinatorial circuits, by definition, do not have state.

Figure 22-30shows a digital logic circuit containing both sequential and combinatorial modules. We can thus model digital circuits as a controlled interaction between a set of sequential and combinatorial circuits. Solving problems using digital circuits requires controlling the flow of data through the circuit in such a way that it provides a solution to the given problem.

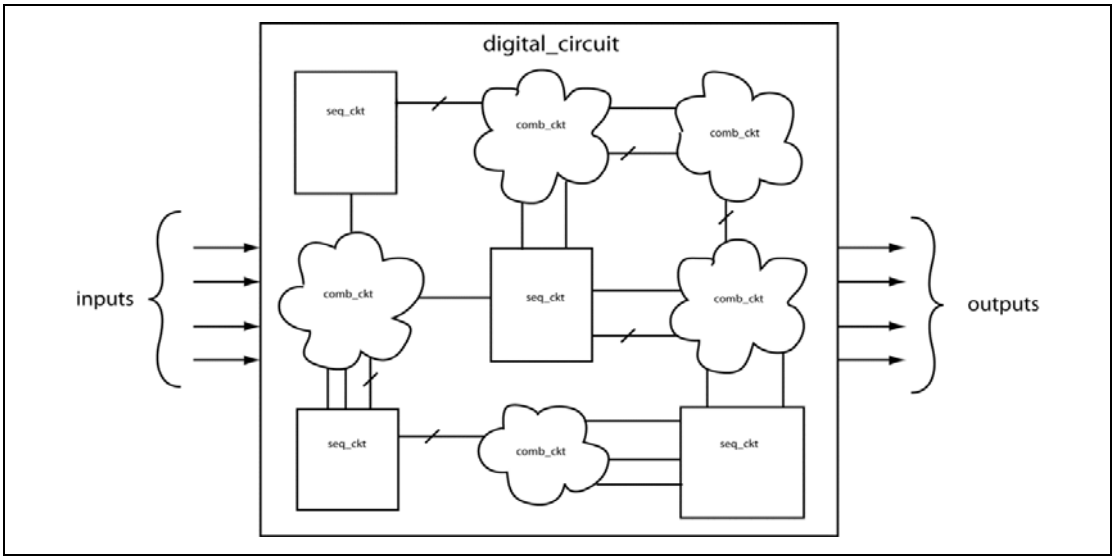


Figure 22-30: A basic logic circuit.

Figure 22-31(a) shows the basic model of a digital logic circuit; we characterize the signals that the outside world sees as either inputs or outputs. Because we need to control the flow of data through the digital circuit, we must more specifically define the inputs and outputs of a basic digital circuit module. Figure 22-31(b) shows that we further classify the inputs as either "data" or "control" and classify the outputs as either "data" or "status". This means the various circuit elements in Figure 22-31(b) are able to 1) pass data from their inputs to their outputs under the direction of the "control" inputs and, 2) output characteristics of the data transfers using the status outputs.

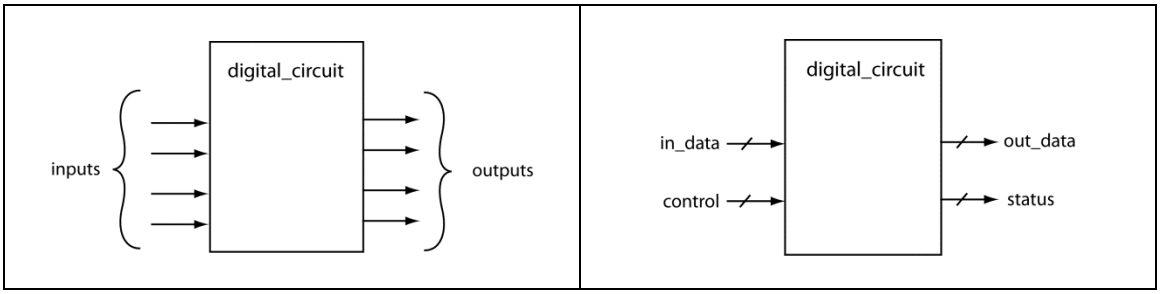


Figure 22-31: Models for a basic logic circuit (a), and a more refined basic digital logic circuit (b).

Something must control the flow of data through the generic digital circuit. We therefore must have some other entity that interprets the status signal outputs of the circuit modules and issues control signals to those circuit modules. For the purpose of this discussion, we'll consider this circuit to be a finite state machine (FSM). The important thing to remember is that something controls the circuit, whether it is an FSM, a computer, or a herd of confused administrators.

Figure 22-32 shows a generic model of an FSM. The FSM simply interprets the status signal outputs from various digital modules and then outputs the appropriate control signals that are the various digital modules use as control inputs. Other interesting characteristics to note include:

- FSMs generally do not have data inputs and data outputs. You can design FSMs with data inputs and outputs, but they tend to be klunky and non-generic. Non-generic FSMs will require modifications if the data widths within the controlled circuit change.

- The FSM is a sequential circuit because it has the ability to store bits. The FSM only stores bits to represent the "state" of the FSM, which it does in its "state variables".

- The underlying model of the FSM includes three primary elements: 1) the next state decoder, 2) the output decoder, and, 3) the state variables. The next state decoder is a combinatorial circuit that decides the next state based on the given state and status inputs. The output decoder is a combinatorial circuit that generates control outputs based on either state only (Moore machine) or state and status inputs (Mealy machine). Figure 22-33 shows models for the Moore and Mealy-type FSMs.

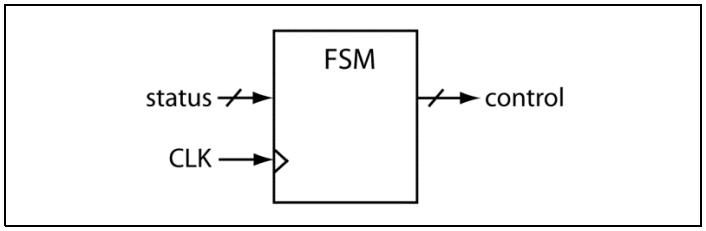


Figure 22-32: A black box model of a FSM.

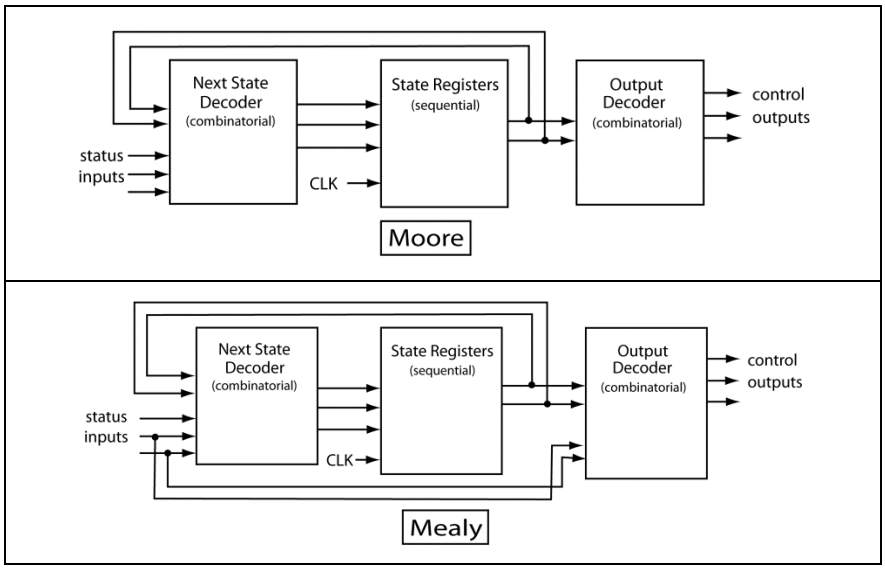


Figure 22-33: The two types of FSMs: (a) Moore, and, (b) Mealy.

Figure 22-34 shows a modified version of Figure 22-31 that includes an FSM as a control element. Figure 22-35 shows that we can further this abstraction. Figure 22-35 shows that the circuit control elements can either be hardware (FSMs) or software (microcontrollers). Additionally, Figure 22-35 shows that the modules that we can control in a digital circuit include "computer peripherals" as well as the low-level digital modules Figure 22-31. Figure 22-35 represents computer peripherals using circles.

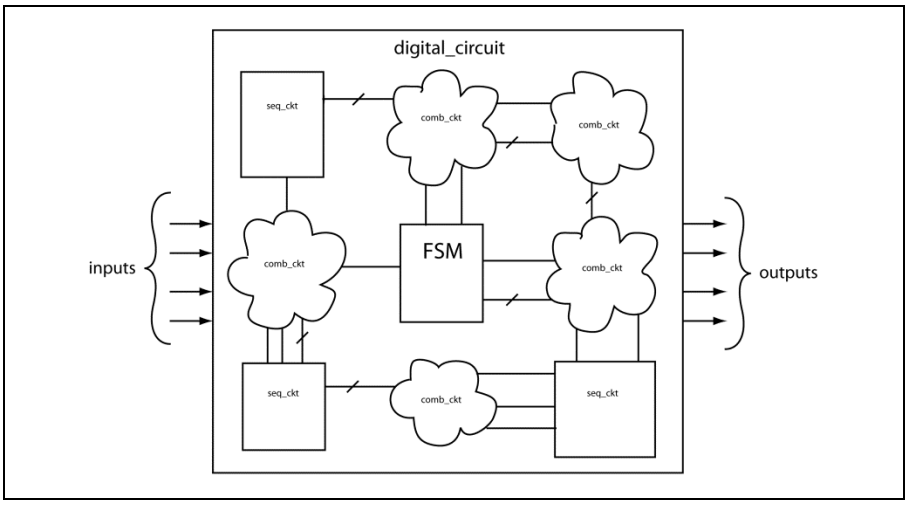


Figure 22-34: A basic logic circuit controlled by FSM

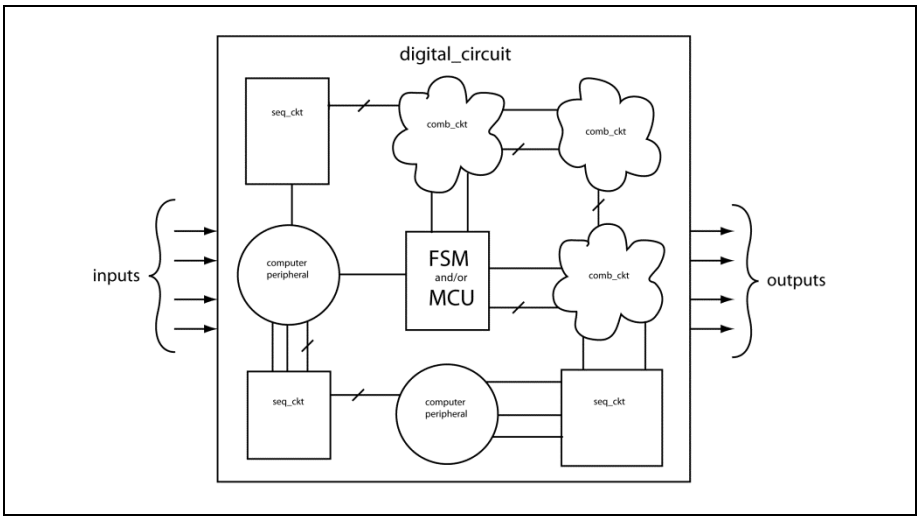


Figure 22-35: A basic logic circuit with peripherals and various control circuits.

Ripple Carry Adder (RCA)

The RCA is a combinatorial module that performs addition; it comprises of a series of Full Adders (FAs) connected in series such that the Co from one module connects to the Cin of the next higher bit location. The RCA can also perform subtraction by changing the sign of one addend before performing the addition.

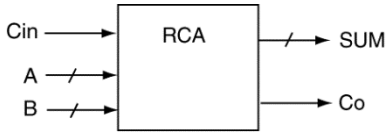
Diagram	Input/Output
	Data In: **A**, **B**, **Cin**. A & B are the addends; Cin is the carry in. Control In: none Data Out: **SUM**. Summation of: A+B+Cin. Status Out: **Co**. The Carry out; indicates if the RCA's addition generated a carry out of the module's MSB.

Figure 22-36: The RCA module overview.

Multiplexor (MUX)

The MUX is a combinatorial circuit that selects which of many (more than one) data inputs appear on the circuit's single data output. The SEL signal determines which signals transfers to the output, which requires that it have a width of at least: $\lceil log_2(\text{number of data inputs}) \rceil$. The width of the data inputs and outputs are equivalent. The most generic forms of MUXes include 2:1, 4:1, 8:1, 16:1, etc.

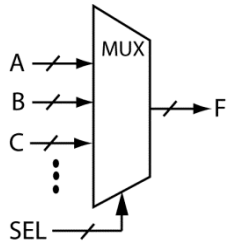
Diagram	Input/Output
	Data In: **A**, **B**, **C, etc**. MUXes can have any number of data inputs. Control In: **SEL**. selects which data input appears on the F**.** The width of the SEL signal is such that $2^{SEL} \geq$ to the number of data inputs. Data Out: **F**. A single output which is equivalent to one of the inputs as selected by the SEL signal. Status Out: none

Figure 22-37: The MUX module overview.

Comparator

The comparator is a combinatorial circuit that generates an equality-type relationship between the two inputs. The comparator has outputs of EQ (equal), LT (less than), and GT (greater than) which are characteristics of the relationship between the two input. The comparator's two input values are typically bundled values of equal width.

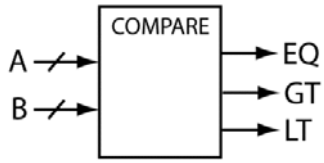
Diagram	Input/Output
	Data In: **A**, **B**. the two bundled values to be compared. Control In: none Data Out: none Status Out. **EQ** (A=B), **LT** (A<B), **GT** (A>B).

Figure 22-38: The Comparator module overview.

Generic Decoder

The generic decoder is a combinatorial circuit that establishes a functional relationship between the module's data inputs and data outputs. The generic decoder is a digital circuit implementation of a look-up-table (LUT). The generic decoder's inputs and outputs are hard to classify because inputs can include data and/or control and outputs can contain at and/or status. We thus describe this circuit using the IN_DATA input for all the inputs (whether they be data or control) and the OUT_DATA for all outputs (whether they be data or status). Both IN_DATA and OUT_DATA can be bundles or single bits, which allows us to include basic logic gates as generic decoders.

Diagram	Input/Output
	Data In: **IV_DATA**; the function's independent variables Control In: none Data Out: **DV_DATA**; the function's dependent variables Status Out: none

Figure 22-39: The Generic Decoder module overview.

Standard Decoder

The standard decoder is a combinatorial and is a subset of generic decoders. The standard decoder has a special relationship between the SEL inputs and the outputs. The number of single-bit outputs = $2^{(width\ of\ SEL)}$. The output bits have either a one-hot (only one output bit is set) or one-cold (only one output bit is cleared) form. We often describe standard decoders using the notation: 1:2, 2:4, 3:8, 4:16, etc.

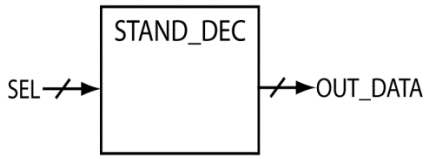
Diagram	Input/Output
	Data In: none. Control In: **SEL**; selects the form of the output. Data Out: **F**; output in one-hot or one-cold form. Status Out: none

Figure 22-40: The Standard Decoder module overview.

Parity Generator

The parity generator is a combinatorial circuit that establishes a given parity for the aggregate combination of the DATA inputs and PAR output. In other words, the parity generator assigns the parity bit such that the DATA & PAR bits are either odd or even parity. Parity "checkers" circuits are similar to parity generators, where the PAR output indicates the DATA bits are either odd or even parity.

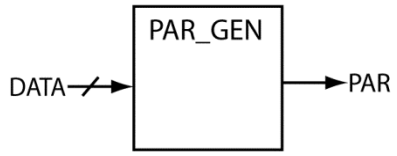
Diagram	Input/Output
	Data In: **IN_DATA.** data used for establishing Control In: none Data Out: none Status Out: **PAR**. the bit that creates the appropriate parity for the **DATA** & **PAR** aggregate value.

Figure 22-41: The Parity Generator module overview.

Registers

The register is a sequential circuit that to stores bits of data. Registers generally store multiple bits of data; we refer to a 1-bit register as a flip-flop. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. Any data loaded to the register appears on the register's OUT_DATA output after a given propagation delay. Registers also have inputs such as CLR, which clears each of the bit storage elements in the register. Signals such as CLR are often asynchronous, which means the given action occurs immediately upon asserting the CLR signal.

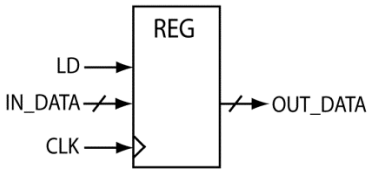
Diagram	Input/Output
	Data In: **IN_DATA**; data to be synchronously loaded into the register. Control In: **CLK, LD, CLR**; The CLK signal synchronizes the loading of data into the register, which happens when both an active clock edge occurs when the LD input is asserted. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data previously loaded to the register. Status Out: none

Figure 22-42: The Register module overview.

Counters

The counter is a sequential circuit that is a special type of register, which means it retains all the control inputs associated with a register. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. The counter as the ability to count up (adds '1' to stored value) or count down (subtracts '1 from stored value). As with the LD signal, changes to the stored register value are synchronized with the active clock edge. Counter typically have inputs such as CLR, which serves to clear each of the bit storage elements in the register. Counter also have inputs such as CLR, which clears each of the bit storage elements in the register. Counters generally automatically "roll over" when they reach their terminal values, which means that the count transitions from its maximum value to zero when counting up and from zero to its maximum value when counting down.

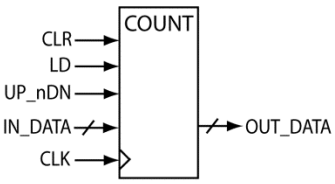
Diagram	Input/Output
	Data In: **IN_DATA**; parallel data for synchronous loading. Control In: **CLK, LD, CLR, UP_NDN, CEN**; The CLK signal synchronizes the loading of data into the register and the changing of the count, which can either be up or down as controlled by the UP_NDN signal. The LD signal controls the loading of new data into the register while the CEN signal enables the synchronous counting of the circuit. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data previously stored in the circuit and possibly modified as loaded to the register. The OUT_DATA signal is the "count" value of the counter. Status Out: **RCO**; establishes when the counter outputs are at the counter's terminal value. The terminal value depends on the count direction (UP_NDN).

Figure 22-43: The Counter module overview.

Shift Registers

The shift register is a sequential circuit that is a special type of register. The register's load control input (LD) enables the register to load the input data to the register; the register synchronizes this loading with the active edge (rising or falling) edge of the CLK signal. The SHR & SHL signals cause synchronous right and left-

shifting of the data in the register, respectively; the in_data bit becomes the new MSB for right shift or the new LSB for left shifts. Shift registers can contain CLR and HOLD inputs which, clears each of the bit storage elements or does not change them, respectively. Shift registers most often bundle the control inputs into a single select-type signal.

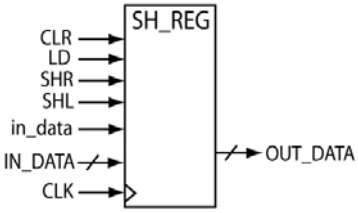
Diagram	Input/Output
	Data In: **IN_DATA, in_data**; IN_DATA is the multibit signal to be loaded into the SR; in_data is the single bit of that becomes the MSB or LSB of the stored value for right or left shifts, respectively. Control In: **CLK, LD, CLR, SHR, SHL, HOLD**; The CLK signal synchronizes the loading of data into the register, which happens when both an active clock edge occurs when the LD input is asserted. The SHR and SHL (shift right & shift left) signals cause the store bits to synchronously shift either right (in_data becomes the new MSB) or left (in_data becomes the new LSB). The HOLD signal prevents changes in the registers content. The CLR input clears each bit storage element in the register either synchronously or asynchronously. Data Out: **OUT_DATA**; the data stored in the shift register. Status Out: none

Figure 22-44: The Counter module overview.

Random Access Memory (RAM)

The RAM is a sequential circuit that allows for the storage of large amounts of data relative to registers. RAM contains three types of input/output signals: address, data (input and output), and control. The IN_DATA signal is the data that will write to the RAM; the OUT_DATA is the data that is read from RAM. All data reads and write occur at the RAM location specified by the address inputs. The value of the control signals allow the data reads or writes. While memory modules often have many control signals, we'll only consider a CLK and a WE (write enable) signal in order to simplify this description. Reading data from the RAM is an asynchronously operation; writing to the RAM is a synchronous operation. Read Only Memories (ROMs) have most of the same features of a RAM except for the WE signal. Additionally, reading from ROMs can either be synchronous or asynchronous.

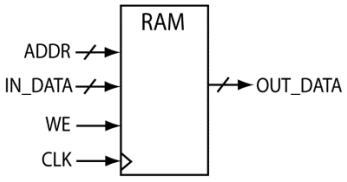
Diagram	Input/Output
	Data In: **IN_DATA**; data to be synchronously written to RAM. Control In: **CLK, WE, ADDR**; The CLK signal synchronizes the writing of data to the RAM; the RAM stores the value of IN_DATA at the address associated with the value of ADDR on the active clock edge (synchronously) when the WE signal is asserted. Data read from RAM is asynchronous; data appearing on OUT_DATA is the data associated with the ADDR value when WE is not asserted. Data Out: **OUT_DATA**; the data currently stored in the shift register. Status Out: none

Figure 22-45: The Counter module overview.

Finite State Machine (FSM)

The FSM is a sequential circuit that controls other digital circuits. The FSM reacts to status inputs and issues appropriate control outputs. The FSM's control inputs are "status" outputs form other digital modules, while the FSM's status outputs become "control" inputs to other digital modules. The FSM uses the value of the status inputs to transition through the various states of the FSM on the active edge of the CLK signal. The control outputs can either Moore (outputs a function of state only) or Mealy-type (outputs are a function of both state and status inputs).

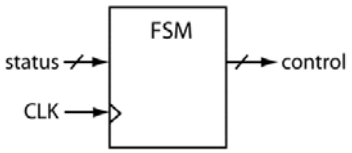
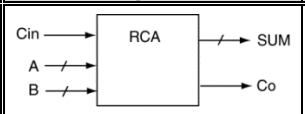
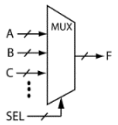
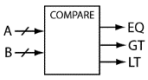
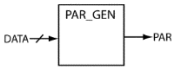
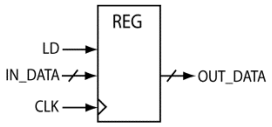
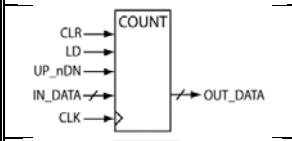
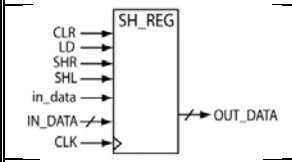
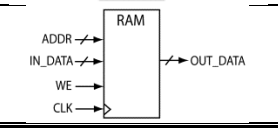
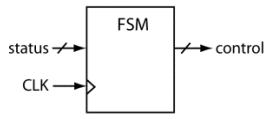
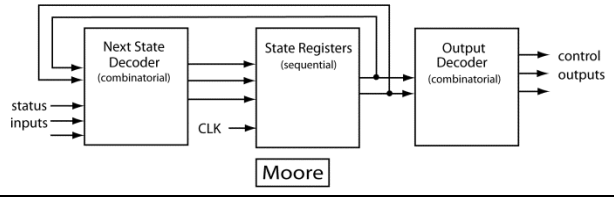
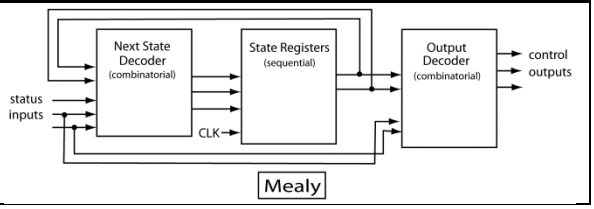
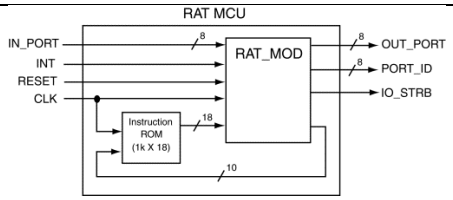
Diagram	Input/Output
	Data In: none Inputs: **CLK, status**; The CLK signal synchronizes state transitions of the FSM; status inputs are the status outputs of modules external to the FSM. Status input values determine the FSM's state transitions. Outputs: **control**; circuit elements external to the FSM use the control outputs to facilitate data handling; control outputs can be either Mealy or Moore-type. Status Out: none

Figure 22-46: The Finite State Machine.

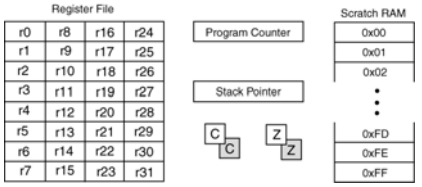
		Circuit Diagram	Data IN	Control IN	Data OUT	Status OUT
CombInatorial	**RCA**		A B Cin	-	SUM	Co
	MUX		Multiple DATA	SEL	Single DATA	-
	Generic Decoder (LUT)		IN_DATA	-	OUT_DATA	-
	Standard Decoder		IN_DATA	SEL	OUT_DATA	-
	Comparator		A B	-	-	EQ GT LT
	Parity Generator		DATA	-	-	PARITY
SequentIal	**Register**		IN_DATA	CLK LD CLR	OUT_DATA	-
	Counter		IN_DATA	CLK LD UP/DOWN ENABLE	COUNT	RCO
	Shift Register		IN_DATA	CLK LD SH LEFT/RIGHT data ENABLE	OUT_DATA	-
	RAM		IN_DATA -	CLK WE ADDR	OUT_DATA -	-
				Inputs		**Outputs**
	FSM		-	CLK status	-	control
FSM Models						

DATA = multiple bits data = single bit

RAT MCU Architecture and Assembly Language Cheat Sheet



schematic diagram



programming model

RAT Instruction Set:

Program Control		Interrupt	Input/Output
`BREQ label`	`BRN label` `CLC`	`RETID`	`IN rX,pp`
`BRNE label`	`SEC`	`RETIE`	`OUT rX,pp`
`BRCS label`	`CALL label`	`SEI`	
`BRCC label`	`RET` `WSP rX`	`CLI`	

Logical		Arithmetic		Shift & Rotate	Storage	
`AND rX,kk`	`AND rX,rY`	`ADD rX,kk`	`ADD rX,rY`	`LSL rX`	`ST rX,imm`	`PUSH rX`
`OR rX,kk`	`OR rX,rY`	`ADDC rX,kk`	`ADDC rX,rY`	`LSR rX`	`ST rX,(rY)`	`POP rX`
`EXOR rX,kk`	`EXOR rX,rY`	`SUB rX,kk`	`SUB rX,rY`	`ROL rX`	`LD rX,imm`	
`TEST rX,kk`	`TEST rX,rY`	`SUBC rX,kk`	`SUBC rX,rY`	`ROR rX`	`LD rX,(rY)`	`MOV rX,rY`
		`CMP rX,kk`	`CMP rX,rY`	`ASR rX`		`MOV rX,imm`

Fun Facts:
- Max program size: 1024 instructions (18-bit/instr)
- 32 8-bit general purpose registers (GPRs)
- Stack: implemented in scratch RAM

I/O:
- Port Mapped device (8-bit port addresses)
- 8-bit GPR-based Input and Output

Interrupt Architecture:
- Interrupt on: interrupt input high voltage & interrupt enabled
- One maskable external interrupt (see interrupt group)
- Vector interrupt: vector address 0x3FF
- Context saving: C & Z flags saved on interrupt
- Interrupt automatically disabled on interrupt
- Context restoration: C & Z flags restored on RETID or RETIE instructions

Bit Masking:

bit setting: OR with '1'	bit clearing: AND with '0'	bit toggling: XOR with '1'
`OR r3,0x01 ;set bit 0`	`AND r3,0xFE ;clear bit 0:`	`EXOR r3,0x01 ;toggle bit 0`

Conditional Constructs:

if/else construct

input byte; if byte is 0x00, then r1=0x0A; otherwise, r1=0x0B output r1

```
            IN      r0,IN_PORT      ; grab data
            CMP     r0,0x00         ; test to see if byte is 0x00
            BRNE    not_zero        ; jump if r0 is not 0x00
            MOV     r1,0x0A         ; place 0x0A in r1
            BRN     out_val         ; jump to output instruction
                                    ;
not_zero:   MOV     r1,0x0B         ; place 0x0B in r1
out_val:    OUT     r1,OUT_PORT     ; output some data
```

iterative construct
(known iterative value)

place iterative value in r3; do something; decrement count; check to see if zero; repeat if not zero

```
            MOV     r3,0x08         ; load iterative count value
loop:       ;                       ; do something meaningful
            ;
            SUB     r3,0x01         ; decrement iteration variable
            BRNE    loop            ; do it again if count non-zero
            ;
loop_done:  ; do something else…
```

conditional construct
(unknown iterative value)

Input value; increment it; do it again if carry flag is not set

```
            MOV     r0,0x00         ; clear register
loop:       IN      r1,IN_PORT      ; grab some data
            ADD     r0,r1           ; add some value to r0
            BRCC    loop            ; repeat if no carry

loop_done:  ; do something else…
```

Equality/Inequality:

- C and Z flag are used to establish relationship between registers:

Operation:		`SUB rA, rB` `CMP rA, rB`
C	**Z**	**Comment**
0	0	rA > rB
1	0	rA < rB
-	1	rA = rB

VHDL Cheat Sheet

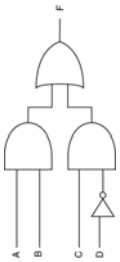
Concurrent Statements		Sequential Statements
Concurrent Signal Assignment (dataflow model)	⇔	**Signal Assignment**
target <= *expression*;		*target* <= *expression*;
A <= B **AND** C; DAT <= (D **AND** E) **OR** (F **AND** G);		A <= B **AND** C; DAT <= (D **AND** E) OR (F **AND** G);
Conditional Signal Assignment (dataflow model)	⇔	***if* statements**

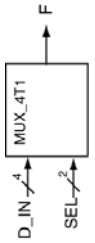
target <= *expressn* **when** *condition* **else**
expressn **when** *condition* **else**
expressn;
		if (*condition*) **then**
		{ sequence of statements }
		elsif (*condition*) **then**
		{ sequence of statements }
		else --(the else is optional)
		{ sequence of statements }
		end if;

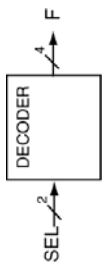
F3 <= '1' **when** (L='0' **AND** M='0') **else** '1' **when** (L='1' **AND** M='1') **else** '0';		**if** (SEL = "111") **then** F_CTRL <= D(7); **elsif** (SEL = "110") **then** F_CTRL <= D(6); **elsif** (SEL = "101") **then** F_CTRL <= D(1); **elsif** (SEL = "000") **then** F_CTRL <= D(0); **else** F_CTRL <= '0'; **end if;**
Selective Signal Assignment (dataflow model)	⇔	***case* statements**
with *chooser_expression* **select** target <= *expression* **when** *choices*, *expression* **when** *choices;*		**case** (*expression*) **is** **when** *choices* => {sequential statements} **when** *choices* => {sequential statements} **when others** => -- (optional) {sequential statements} **end case**;
with SEL **select** MX_OUT <= D3 **when** "11", D2 **when** "10", D1 **when** "01", D0 **when** "00", '0' **when others**;		**case** ABC **is** **when** "100" **=>** F_OUT **<=** '1'; **when** "011" **=>** F_OUT **<=** '1'; **when** "111" **=>** F_OUT **<=** '1'; **when others =>** F_OUT **<=** '0'; **end case;**
Process (behavioral model)		
opt_label: **process**(*sensitivity_list*) **begin** {*sequential_statements*} **end process** *opt_label*;		

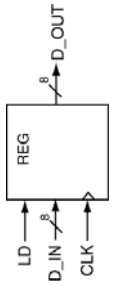
proc1: **process**(A,B,C)
begin
if (A = '1' **and** B = '0') **then**
F_OUT **<=** '1';
elsif (B = '1' **and** C = '1') **then**
F_OUT **<=** '1';
else
F_OUT **<=** '0';
end if;
end process proc1;		

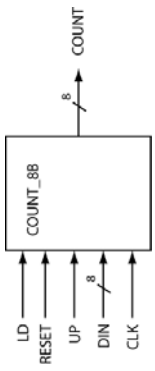
Description	CKT Diagram	VHDL Model

Typical logic circuit		`entity my_ckt is`
		`  Port ( A,B,C,D : in std_logic;`
		`  F : out std_logic);`
		`end my_ckt;`
		
		`architecture ckt1 of my_ckt is`
		`begin`
		`  F <= (A AND B) OR (C AND (NOT D));`
		`end ckt1;`
		
		`architecture ckt2 of my_ckt is`
		`begin`
		`  F <= '1' when (A = '1' AND B = '1') else`
		`  '1' when (C = '1' AND D = '0') else`
		`  '0';`
		`end ckt2;`

4:1 Multiplexor		`entity MUX_4T1 is`
		`  Port (  SEL : in std_logic_vector(1 downto 0);`
		`  D_IN : in std_logic_vector(3 downto 0);`
		`  F : out std_logic);`
		`end MUX_4T1;`
		
		`architecture my_mux of MUX_4T1 is`
		`begin`
		`  F <= D_IN(0) when (SEL = "00") else`
		`  D_IN(1) when (SEL = "01") else`
		`  D_IN(2) when (SEL = "10") else`
		`  D_IN(3) when (SEL = "11") else`
		`  '0';`
		`end my_mux;`

2:4 Decoder		`entity DECODER is`
		`  Port ( SEL : in std_logic_vector(1 downto 0);`
		`  F : out std_logic_vector(3 downto 0));`
		`end DECODER;`
		
		`architecture my_dec of DECODER is`
		`begin`
		`  with SEL select`
		`  F <= "0001" when "00",`
		`  "0010" when "01",`
		`  "0100" when "10",`
		`  "1000" when "11",`
		`  "0000" when others;`
		`end my_dec;`

8-bit register with load enable		`entity REG is`
		`  port ( LD,CLK : in std_logic;`
		`  D_IN : in std_logic_vector (7 downto 0);`
		`  D_OUT : out std_logic_vector (7 downto 0));`
		`end REG;`
		
		`architecture my_reg of REG is`
		`begin`
		`  process (CLK,LD)`
		`  begin`
		`  if (LD = '1' and rising_edge(CLK)) then`
		`  D_OUT <= D_IN;`
		`  end if;`
		`  end process;`
		`end my_reg;`

8-bit up/down counter with asynchronous reset		`entity COUNT_8B is`
		`  port ( RESET,CLK,LD,UP : in std_logic;`
		`  DIN : in std_logic_vector (7 downto 0);`
		`  COUNT : out std_logic_vector (7 downto 0));`
		`end COUNT_8B;`
		
		`architecture my_count of COUNT_8B is`
		`  signal  t_cnt : std_logic_vector(7 downto 0);`
		`begin`
		`  process (CLK, RESET)`
		`  begin`
		`  if (RESET = '1') then`
		`  t_cnt <= (others => '0'); -- clear`
		`  elsif (rising_edge(CLK)) then`
		`  if (LD = '1') then  t_cnt <= DIN;  -- load`
		`  else`
		`  if (UP = '1') then  t_cnt <= t_cnt + 1; -- incr`
		`  else  t_cnt <= t_cnt - 1; -- decr`
		`  end if;`
		`  end if;`
		`  end if;`
		`  end process;`
		`  COUNT <= t_cnt;`
		`end my_count;`

Finite State Machine Modeling using VHDL Behavioral Models

```
entity fsm is
    port ( X,CLK,RESET : in  std_logic;
                     Y : out std_logic_vector(1 downto 0);
                 Z1,Z2 : out std_logic);
end fsm;

architecture my_fsm of fsm is
   type state_type is (ST0,ST1,ST2,ST3);
   signal PS,NS : state_type;
begin

sync_proc: process(CLK,NS,RESET)
   begin
     if (RESET = '1') then PS <= ST0;
     elsif (rising_edge(CLK)) then PS <= NS;
     end if;
   end process sync_proc;


comb_proc: process(PS,X)
   begin
      Z1 <= '0'; Z2 <= '0';
      case PS is
         when ST0 =>    -- items regarding state ST0
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST1; Z2 <= '0';
            else  NS <= ST0; Z2 <= '1';
            end if;
         when ST1 =>    -- items regarding state ST1
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST2; Z2 <= '0';
            else  NS <= ST1; Z2 <= '1';
            end if;
         when ST2 =>    -- items regarding state ST2
            Z1 <= '0';  -- Moore output
            if (X = '0') then NS <= ST3; Z2 <= '0';
            else  NS <= ST2; Z2 <= '1';
            end if;
         when ST3 =>    -- items regarding state ST3
            Z1 <= '1';  -- Moore output
            if (X = '0') then NS <= ST0; Z2 <= '0';
            else  NS <= ST3;  Z2 <= '1';
            end if;
         when others => -- the catch all default case
            NS <= ST0; Z1 <= '0'; Z2 <= '0';
      end case;
   end process comb_proc;

with PS select
   Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
end my_fsm;
```

Description of FSM's interface (I/O)

Declaration of VHDL type used to represent the states of the FSM.

The "synchronous" process to control FSM parameters associated with the state variable representation (storage elements).

The combinatorial process to handles output decoding and next-state assignments.

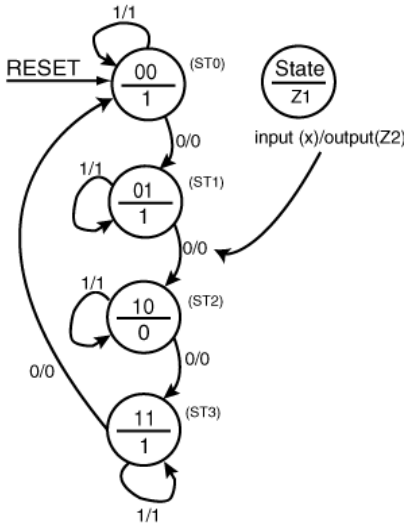
All outputs are assigned initial values at start of process.

The "cases": one case for each FSM state (and a default case too).

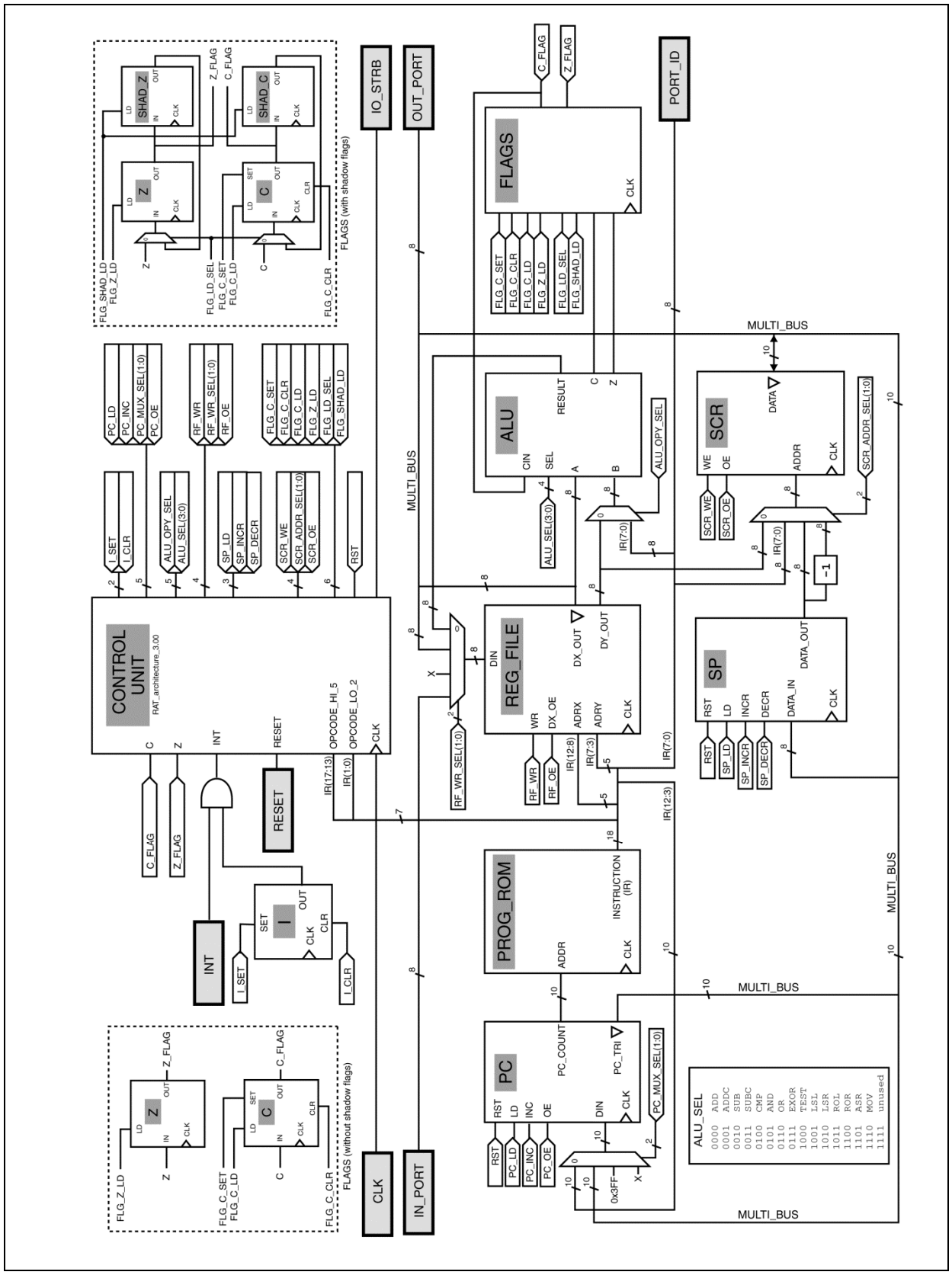
Moore output assigments are outside conditional statement

Mealy output assignments and next state assigments are inside the conditional statement.

Concurrent statement to provide desired output assignments based on FSM states; required if the state variables are used as FSM outputs.



RAT MCU Architecture Diagrams



RAT MCU Wrapper Source Code

```
----------------------------------------------------------------------------------
-- Company:  RAT Technologies
-- Engineer:  Various RAT rats
--
-- Create Date:    1/31/2012
-- Design Name:
-- Module Name:    RAT_wrapper - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description: Wrapper for RAT MCU. This model provides a template to
--    interface the RAT MCU to the development board.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RAT_wrapper is
    Port ( LEDS     : out   STD_LOGIC_VECTOR (7 downto 0);
           SEGMENTS : out   STD_LOGIC_VECTOR (7 downto 0);
           DISP_EN  : out   STD_LOGIC_VECTOR (3 downto 0);
           SWITCHES : in    STD_LOGIC_VECTOR (7 downto 0);
           BUTTONS  : in    STD_LOGIC_VECTOR (3 downto 0);
           RESET    : in    STD_LOGIC;
           CLK      : in    STD_LOGIC);
end RAT_wrapper;

architecture Behavioral of RAT_wrapper is

   ---------------------------------------------------------------------------------
   -- INPUT PORT IDS ---------------------------------------------------------------
   CONSTANT SWITCHES_ID : STD_LOGIC_VECTOR (7 downto 0) := X"20";
   CONSTANT BUTTONS_ID  : STD_LOGIC_VECTOR (7 downto 0) := X"24";
   ---------------------------------------------------------------------------------

   ---------------------------------------------------------------------------------
   -- OUTPUT PORT IDS --------------------------------------------------------------
   CONSTANT LEDS_ID       : STD_LOGIC_VECTOR (7 downto 0) := X"40";
   CONSTANT SEGMENTS_ID   : STD_LOGIC_VECTOR (7 downto 0) := X"82";
   CONSTANT DISP_EN_ID    : STD_LOGIC_VECTOR (7 downto 0) := X"83";
   ---------------------------------------------------------------------------------

   ---------------------------------------------------------------------------------
   -- Declare RAT_MCU --------------------------------------------------------------
   component RAT_MCU
       Port ( IN_PORT  : in  STD_LOGIC_VECTOR (7 downto 0);
              OUT_PORT : out STD_LOGIC_VECTOR (7 downto 0);
              PORT_ID  : out STD_LOGIC_VECTOR (7 downto 0);
              IO_STRB  : out STD_LOGIC;
              RESET    : in  STD_LOGIC;
              INT      : in  STD_LOGIC;
              CLK      : in  STD_LOGIC);
   end component RAT_MCU;
   ---------------------------------------------------------------------------------

   -- Signals for connecting RAT_MCU to RAT_wrapper -------------------------------
   signal s_input_port  : std_logic_vector (7 downto 0);
   signal s_output_port : std_logic_vector (7 downto 0);
   signal s_port_id     : std_logic_vector (7 downto 0);
   signal s_load        : std_logic;
```

```
   --signal s_interrupt   : std_logic; -- not yet used

   -- Register definitions for output devices ----------------------------------
   signal r_LEDS        : std_logic_vector (7 downto 0) := (others => '0');
   signal r_SEGMENTS    : std_logic_vector (7 downto 0) := (others => '0');
   signal r_DISP_EN     : std_logic_vector (3 downto 0) := (others => '0');
   ------------------------------------------------------------------------------

begin

   -- Instantiate RAT_MCU ------------------------------------------------------
   MCU: RAT_MCU
   port map(  IN_PORT  => s_input_port,
              OUT_PORT => s_output_port,
              PORT_ID  => s_port_id,
              RESET    => RESET,
              IO_STRB  => s_load,
              INT      => '0',
              CLK      => CLK);
   ------------------------------------------------------------------------------


   ------------------------------------------------------------------------------
   -- MUX for selecting what input to read ------------------------------------
   ------------------------------------------------------------------------------
   inputs: process(s_port_id, SWITCHES)
   begin
      if (s_port_id = SWITCHES_ID) then
         s_input_port <= SWITCHES;
      elsif (s_port_id = BUTTONS_ID) then
         s_input_port <= BUTTONS;
      else
         s_input_port <= x"00";
      end if;
   end process inputs;
   ------------------------------------------------------------------------------

   ------------------------------------------------------------------------------
   -- MUX for updating output registers ---------------------------------------
   -- Register updates depend on rising clock edge and asserted load signal
   ------------------------------------------------------------------------------
   outputs: process(CLK)
   begin
      if (rising_edge(CLK)) then
         if (s_load = '1') then

            -- the register definition for the LEDS
            if (s_port_id = LEDS_ID) then
               r_LEDS <= s_output_port;
            elsif (s_port_id = SEGMENTS_ID) then
               r_SEGMENTS <= s_output_port;
            elsif (s_port_id = DISP_EN_ID) then
               r_DISP_EN <= s_output_port(3 downto 0);
            end if;

         end if;
      end if;
   end process outputs;
   ------------------------------------------------------------------------------

   -- Register Interface Assignments -------------------------------------------
   LEDS <= r_LEDS;
   SEGMENTS <= r_SEGMENTS;
   DISP_EN <= r_DISP_EN;
   ------------------------------------------------------------------------------

end Behavioral;
```

VHDL Style File

The main goal of your VHDL source code is to model a digital circuit. This means that your only mission is to satisfy the VHDL synthesizer. But in reality, you and other humans are going to need to read and understand your VHDL source code. The single most important factor in creating readable and understandable VHDL source code is to make sure your VHDL source code follows certain appearance guidelines. The file in this section shows some more of the more important attributes and rules that all VHDL source code should adhere to. The overriding factor with your VHDL source code is to make it neat, organized, and readable; any specific items not listed in this style files should adhere to these principles.

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer: Manuel Laybor, Ima Goodstud
--
-- Create Date: 09/10/2007
-- Design Name: VHDL Style File
-- Module Name: comp2
-- Project Name: style file example
-- Target Devices: Digilent Nexys Board
-- Tool versions:
-- Description: Describe the purpose of this file. This should be a high-level
--     description but it should be complete. The low-level implementation
--     details should appear in the body of the VHDL code in the form of comments.
--
-- Dependencies: If the synthesis of this file depends on other file, be
--     sure to list them here.
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments: Say stuff that may be helpful to others who may be
--     wanting to use this file for their own projects.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- delete meaningless comments (such as this one)

-- entity declaration should appear neat; much of the declaration
--  is aligned to increase readability.
entity comp2 is
    Port ( A,B : in  STD_LOGIC_VECTOR (9 downto 0);
           EQ : out STD_LOGIC);
end comp2;

-- code is indented;
--  comments align with code indentations
architecture my_comp2 of comp2 is
begin
   -- all items in architecture body are indented

   -- descriptive label is included with process statements
   my_comp: process(A,B)
   begin
      if (A = B) then
         EQ <= '1';
      else
         EQ <= '0';
      end if;
   end process my_comp;

end my_comp2;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity comp3
    Port ( A,B,C : in  STD_LOGIC_VECTOR (9 downto 0);
           EQ3 : out STD_LOGIC);
end comp3;

architecture my_comp3 of comp3 is
   -- all items in declarative region of architecture are indented

   -- component declaration
   component comp2
       Port ( A,B : in  STD_LOGIC_VECTOR (9 downto 0);
              EQ : out STD_LOGIC);
   end component;

   -- intermediate signals: note proper prefixing
   signal s_eq1, s_eq2 : std_logic;

begin
   -- blank lines are used to delineate instantiations
   --  and signal assignment statements

   -- component instantiations:
   --  label and entity name one first line
   --  port mappings on subsequent lines
   --
   --  port mappings are neat and aligned
   eq1: comp2
   port map (A => A,
             B => B,
             EQ => s_eq1);

   -- component instantiation
   --  another form of alignment used
   eq2: comp2
   port map ( A => A,
              B => C,
             EQ => s_eq2);

   -- basic logic for final output
   EQ3 <= s_eq1 AND s_eq2;

end;
```

RAT MCU Assembly Language Style File

The following file shows some of the more important issues regarding generating neat and readable RAT MCU assembly source code. No style file can show you everything and they rarely make such an attempt. The underlying factor in writing any source code is to be neat and consistent. Using proper indentation, white space and commenting helps you attain the goals of being neat and consistent. The code in the following block is a modified fragment from another working program; if it were to actually assemble, it would not do anything useful. The program is presented primarily for appearance purposes.

```
;-------------------------------------------------------------------
;- Programmer: Pat Wankaholic
;- Date: 09-29-10
;- Experiment #??
;-
;- This program does something really cool. Here's the description...
;-------------------------------------------------------------------

;-------------------------------------------------------------------
;- I/O Constants
;-------------------------------------------------------------------
.EQU SWITCH_PORT = 0x30          ; port for switches ----- INPUT
.EQU LED_PORT    = 0x0C          ; port for LED output --- OUTOUT
.EQU BTN_PORT    = 0x10          ; port for button input - INPUT
;-------------------------------------------------------------------

;-------------------------------------------------------------------
;- Misc Constants
;-------------------------------------------------------------------
.EQU BTN2_MASK = 0x08            ; mask all but BTN5
.EQU B0_MASK   = 0x01            ; mask all but bit0
;-------------------------------------------------------------------

;-------------------------------------------------------------------
;- Memory Designation Constants
;-------------------------------------------------------------------
.DSEG
.ORG     0x00

COW:   .DB 0x09,0x07,0x06,0x05
;-------------------------------------------------------------------
;
.CSEG
.ORG         0x01

init:        SEI                      ; enable interrupts

main_loop:   IN      r0, BTN_PORT     ; input status of buttons
             AND     r0, BTN2_MASK    ; clear all but BTN2
             BRN     bit_wank         ; jumps when BTN2 is pressed

             ;-----------------------------------------------------
             ;- nibble wank portion of code
             ;-----------------------------------------------------
wank:        ROL     r1               ; rotate 2 times - msb-->lsb
bit3:        BRN     fin_out          ; jump unconditionally to led output
             ;-----------------------------------------------------

             ;-----------------------------------------------------
             ; bit-wank algo: do something Blah, blah, blah ...
             ;-----------------------------------------------------
bit_wank:    LD      r0,0x00          ; clear r0 for working register
```

```
            OR      r0, B1_MASK       ; set bit1
bit2:       LSR     r1                ; shift msb into carry bit
            BRCS    bit3              ; jump if carry not set
            ;-------------------------------------------------------------

            CALL    My_sub            ; subroutine call
fin_out:    OUT     r0,LED_PORT       ; output data to LEDs
            BRN     main_loop         ; endless loop
            ;-------------------------------------------------------------


;-------------------------------------------------------------
; Subroutine: My_sub:
;
; This routines does something useful. It expects to find
; some special data in registers r0, r1, and r2. It changes
; the contents of registers blah, blah, blah...
;
; Tweaked registers: r1
;-------------------------------------------------------------
My_sub:     LSR     r1                ; shift msb into carry bit
            BRCS    bit3              ; jump if carry not set
            RET
;-------------------------------------------------------------
```

Computer Design Dictionary

-A-

Academonic: The rallying cry for those who dare to expose the endemic correction in academia.

Arithmetic Logic Unit (ALU): The ALU is generally a datapath submodule, which in turn is a submodule of CPU. The ALU is responsible for standard bit operations such as arithmetic and logical operations (and shifts and any other way you can think of to tweak bits). The ALU is responsible for generating status of various operations (zero, negative, overflow, carry, pointlessness, parity, etc.) which are typically individual bits that are latched outside of the ALU.

Arithmetic shifts: Shift operations that protect the sign of data residing in a shift register when performing shift operations.

Assembler Directives: One of the three main parts of an assembly language program. Assembler directives provide a method for the programmer to send messages to the assembler.

Assembler: An assembler is a computer program that translates assembly code (instruction mnemonics) into machine code.

Assembly Language Program Parts: There are generally three types of information found in assembly language programs: 1) comments, 2) assembler directives, and, 3) assembly language instructions.

Assembly Language: A computer language that uses mnemonics to represent the instructions available to the programmer (the instruction set) for a given computer architecture. The mnemonics roughly spell out what the instruction does in terms of the underlying hardware. Assembly language programs are translated to machine code by use of a software program referred to as an assembler. Assembly language is generally non-portable in that the assembly instructions are specific to a given computer architecture.

-B-

Background Task: A term used to describe the program code associated with interrupt service routines.

Barrel shifts: A special type of shift register shift that shifts any number of bits (other than one bit) on a single clock cycle.

Bi-Directional Signals: A term that refers to the notion that data can flow through a line in two directions (though not at the same time) rather than only one direction. Bi-directions signals are generally associated with tri-state outputs because a given device cannot generally simultaneously drive a signal and read from that signal.

Bit Masks: The term bit-mask describes a value that "selects" certain bit locations of a given word while disregarding other bit locations. The disregarded bits are generally cleared by the bit-masking operation. Bit masking is generally required because most operations in microcontrollers occur on the byte-level.

Bit-Banging: The process of using microcontroller outputs on a bit-level to control external peripherals. In this way, the general purpose outputs of a microcontroller are used to generate the control signals required to control and/or exchange information with external devices.

Block Diagram: A modeling approach used in hardware to quickly transfer high-level knowledge regarding the operations of a given circuit to the human reader. Block diagrams can and should be hierarchical in nature when appropriate to expedite their understanding to the human reader.

Bus Contention: Bus contention occurs when two different busses attempt to simultaneously drive the same bus. In this context, the bus is a *shared resource*. Contention can also occur on individual signals as well as busses.

Bus: A set of electrical signals that are grouped together because they share a common purpose. The term "bus" also refers to various standard data transmission protocols, and as a result, a bus, as defined here is often referred to as a *bundle*.

-C-

Central Processing Unit (CPU): The CPU is generally considered the part of the computer that executes the instructions. Typical submodules of the CPU include the control unit, datapath, program counter, instruction memory, register files, accumulators, ALUs, secondary memory, roach motels, etc.

CISC: This acronym officially stands for "Complex Instruction Set Architecture" and is generally used to describe computer architectures. CISC computers generally have the following characteristics:
The architecture contains relatively few general purpose registers

- The instruction word formats are of different lengths
- Instructions require a different number of clock cycles to complete execution
- Some instructions in the instruction set are complex (meaning they can generate a significant amount of processing internal to the architecture)
- System clock rates are generally slower than their RISC counter-parts.

Combinatorial vs. Sequential Circuits: The outputs of a combinatorial circuit are a function of the current inputs while the outputs of a sequential circuit are a function of the combination of past inputs. Stated differently, combinatorial circuits do not have the ability to "remember" bits while sequential circuits are able to store values and are this considered to have memory.

Compiler: A computer program that translates higher-level language code into machine code. Compilers generally also produce assembly language code listings which are specific to the target computer.

Computer I/O: One of the three main subsections of a computer that allows the computer to interact with the outside world.

Context Restoration: A term describing what a CPU does upon completion of servicing an interrupt. In this case, context restoration refers to the notion that the CPU must return to the state it was in (flags, registers, etc.) before the CPU executed the interrupt service routine.

Context Saving: A term that describes what a CPU must do when an interrupt is acted upon. The general notion is that interrupts are asynchronous and can occur while the CPU is executing some important piece of code. In this case, the CPU will save the current state of the CPU (flags, registers, etc.) before processing executing the interrupt service routine.

-D-

Datapath: The hardware module that is generally considered to do the number crunching associated with instructions. Submodules of the datapath generally include the ALU, register file, accumulator, various selection logic, etc.

Display Multiplexing: An approach typically used by LED-based 7-segment displays that allows the driving device to control many digits without dedicating a signal to each LED in each segment. The general approach is to connect each type of segments with one signal and give each individual display an on/off control. Using this configuration, display multiplexing only actuates one display at a time, but does so at a rate that makes it appear as if all displays are on at the same time. Multiplexing works for humans because of the notion of retinal persistence.

-E-

Elementary Operation: A basic operation performed by a sequential circuit. Elementary operations are most often spoken of in terms of registers. Typical operations performed by registers include loading (generally a parallel load), setting (sets all bits in register), clearing (clears all bits in register), shifting/rotating (specifically for shift registers), and incrementing/decrementing (generally for counters).

-F-

Field Programmable Gate Array (FPGA): A programmable logic device (PLD) is an integrated circuit that contains internal devices that can be configured (or programmed) to implement a given digital circuit. The internal devices include logic, memory, routing, and input/output resources.

Finite State Machine (FSM): An abstract machine that defines a finite set of states, actions performed in those states, and a set of rules defining how the machine transitions from state to state. FSM are generally classified as either *Mealy* or *Moore* machines. FSMs are one of two major hardware devices that are typically used to control other hardware entities. In these cases, FSM inputs are considered status inputs while FSM outputs are considered control outputs.

Firmware: Firmware is a computer program that is written to run on a specific piece of hardware and is thus often associated with embedded systems. Firmware does not refer to the language-level in which the program is written thus can be written in machine code, assembly code, or a higher-level language.

First Five Things for a New CPU: When you first examine a new CPU, the five things you should initially examine are 1) the programmer's model, 2) the instruction set, 3) the interrupt architecture, 4) the memory model, and 5) the I/O architecture.

Flicker: An issue associated with display multiplexing where the multiplexing rate is slow enough for humans to note that displays are not "always on".

Flowchart: A diagram that uses a few distinctive symbols to model the program flow associated with an algorithm. Computer programmers use flowcharts as an aid to program design and/or documentation support. Flowcharts can and should be hierarchical in nature when appropriate. The hardware analogy to a flowchart is the black-box diagram.

Foreground Task: A term used to describe the program code associated with the main loop in a program. The foreground task is generally all the code that is not initialization code or interrupt service routine code.

Fragile: A label attached to code that is unmaintainable. Fragile code breaks if you attempt to modify it, hence the name fragile. The roots of fragile code are a complete lack of planning of the code as well as modifications made by people who don't know what the f**k their doing.

-G-

Generic Decoder: A generic decoder is a hardware implementation of a look-up-table (LUT). LUTs generally establish a functional relationship between inputs and outputs by assigning an output for every unique input.

Ghosting: An issue associated with display multiplexing where an LED is on when it should be off resulting in dimly lit LED showing incorrect information.

-H-

Harvard Architecture: A computer architecture that has separate memory space for both data and instructions.

HDLs (Hardware Description Languages): Text-based languages used to model digital circuits. The main flavors of HDLs include VHDL and Verilog.

Higher-Level Computer Language: A computer language that uses opened-ended expressions and functions to generate desired results. Higher-level languages are generally input to computer programs such as compilers which translate the languages to both assembly code and machine code associated with the target machine. Higher-level languages are generally portable (processor independent) and thus various higher-level language code can be compiled to run on different target machines.

High-Impedance: A term that refers to a device that effectively removes itself from a circuit by turning off its drive current. A device that cannot drive a circuit can no longer affect the circuit and is thus effectively not in the circuit.

-I-

Instruction format: The bit-level description of instructions associated with assembly language instructions. Each instruction is comprised of op-codes in every case, but also can include field codes in most cases.

Instruction Set: The instruction set describes the operations that the computer hardware can perform under program control (either software of firmware).

Interrupt Architecture: A common term used to describe all the characteristics (both hardware and software considerations) of interrupts for a given processor. Every computer device generally has a different interrupt architecture and is thus one of the three important aspects of any computer device (the programmer's model and instruction set are the two other important aspects).

Interrupt Cycle: The steps a CPU goes through in order to handle an interrupt. The interrupt cycle is generally different from "normal" processing cycles.

Interrupt Driven I/O: Another form of I/O characterized by an external device having the ability to change the normal flow of a program by executing a special set of code referred to as the interrupt service routine.

Interrupt Masking: This refers to the notion that most interrupt architectures allow for the prevention of response to interrupts based on software control. Processor support for interrupts generally includes instructions that allow for processor response to interrupt signals (unmasking) or prevent system response to interrupt signals (masking).

Interrupt Service Routine (ISR): When a processor responds to an interrupt, the given interrupt architecture responds by executing a set of instructions known as an interrupt service routine. The ISR is nothing more than a subroutine that is executed after being "called" by some device. ISRs are often referred to as "interrupt handlers".

Interrupt Service Routine: A section of code that the CPU executes automatically as a result of acting on a interrupt.

Interrupt Vector Address: The address the CPU places into the program counter when the CPU acts on an interrupt. Thus, when an interrupt is processed, the first instruction executed is the one residing at the vector address. The instruction at the vector address is generally a branch to the interrupt service routine.

Interrupts: An asynchronous signal from an external device to the processor. Exactly how the processor reacts when an interrupt is received is based on the interrupt architecture for a given processor. In simple terms, an interrupt can be considered a method for internal hardware or external devices to call a special subroutine (ISR). Interrupts signals are generally considered asynchronous in nature which makes them vital to real-time embedded systems.

-J-

-K-

-L-

Latches vs. Flip-flops: Both latches and flip-flops are 1-bit storage devices. Latches are considered to be level-sensitive devices and its outputs can change anytime a change in its inputs occurs. Flip-flops are considered edge-sensitive devices and changes in outputs are synchronized to an edge-sensitive input which is often assigned as a clock signal.

Look-Up Table (LUT): A LUT is a programming or hardware construct that translates an input value to a specific output value. Hardware LUTs are typically implemented with *generic decoders* while software LUTs are generally organized as a list of entries in successive memory locations. Hardware LUTs generally save on logic generation and can be used to speed-up hardware operations. Software/Firmware LUTs are typically used to avoid costly calculations at the cost of dedicating memory resources to the LUT.

-M-

Machine Code: A computer program in its lowest-level form. Machine code is comprised of the 1's and 0's that the computer hardware interprets to perform the given operations specified by the program. Machine code is the only level of programming that hardware can actually understand.

Main Computer Components: The three main components of a computer include: 1) CPU, 2) the I/O, and 3) memory. The CPU is the brains/number crunching portion of the computer, the I/O allows the computer to interact with the outside world, and the memory is generally used for program and intermediate data storage.

Main Types of Code in Assembly Language Programs: There are generally three main part of an assembly language program: 1) initialization code, 2) main task code, and 3) interrupt service routine code.

Mealy vs. Moore FSM Models: There are two classes of finite state machine model which are referred to as Mealy and Moore "machines", or "models". The external outputs of a Moore machine are a function of state only and output changes are thus considered to be synchronized to state changes in the FSM. The external outputs of a Mealy machine are a function of both FSM state and the internal inputs. Changes in external outputs of a Mealy machine are not necessarily synchronized to the changes in FSM state since they are also a function of external inputs.

Memory Bandwidth: Memory bandwidth refers to the amount of data that can be transferred to and from memory. The speed of memory reads and writes are constrained by physical attributes of the device as well as the system in which the device operates in which thusly allow for a maximum amount of information to be transferred to and from the device.

Memory Capacity: The amount of storage a given memory contains. Memory capacity is stated in various forms such as total number of bits, total number of bytes, or total number of words.

Memory Configurations: This term refers to the notion that multiple memories can be configured in ways to obtain different memory capacities (number of accessible storage elements) and different storage characteristics (the width or word-length) of each storage element.

Memory Levels: A term that encompasses the various types of memory in a given system. Generally speaking, the lower-level memories are faster but more expensive than higher-level memories. Computer system deal with a trade-off between program execution speed and expense.

Memory Model: A term that describes the general way a given CPU utilizes the memory resources it has at its disposal.

Memory Performance Measures: Because systems rely heavily on memory, items such as read access times, write cycle times, and memory bandwidth are used to measure the specific performance of memory devices within the system.

Memory Reading: An operation that accesses the contents of memory without changing those contents.

Memory Writing: An operation that changes the contents of memory.

Metastability: Digital circuits can become metastable when a set-up and/or hold time is not met. Metastability is a loose definition and means the circuit's output is neither high nor low and may remain in that state there for an unstated amount of time.

Microoperations: A microoperation is an elementary operation performed on data stored in a register. Microoperations can also include interactions with other registers such as storing the result of microoperations associated with other circuit elements. Microoperations are commonly used in higher-level descriptions of digital circuitry such as computers.

Mnemonic: A set of letters that represents a given operation. Generally speaking, mnemonics loosely describe, in an abbreviated manner, the operation they represent.

Model: A model is a representation of something. A more (definitive) descriptive description of a model is a description of something in terms that highlights the relevant information in that thing while hiding the less useful information. The purpose of a model is to quickly transfer important information to the entity reading the model (whether human, or computer, or member of the EE Faculty). Generally speaking, the quality of any model is determined by its ability to transfer information to the user.

Mono-Stable Multivibrator: A device that has one stable state; the stable state can either be the '0' or '1' state. The device's output is only in the non-stable state momentarily before transitioning to the stable state. This term is a fancy name for a device commonly referred to as a "one-shot"

-N-

Narcissistic Personality Disorder (NPD): A disorder inflicting most faculty members in academia. This disorder is required in order to move up any academic ladder.

-O-

One-Shot: The common name for a mono-stable multivibrator. One-shots are used to synthesize fixed-length signals in response to signal events such as clock edges.

Op-code: A term that is short-hand for "operational code". Op-codes are the bits of an instruction that are used by the control unit to decode which instruction is being executed.

-P-

Pig: A term that completely describes academic administrators.

Polling: Processors use polling to interface with external devices where the process constantly evaluates the status of the external device in order to determine if the device is in need of services from the processor. Polling is considered to be used in "programmed I/O" and is one of three major types of computer related I/O. Polling is generally associated with inefficient embedded system design in that the system is considered to have low overall throughput when executing a polling loop

Pop: An operation associated with stacks where an item is removed from a stack; the stack pointer is appropriately adjusted.

Princeton Architecture: A computer architecture where data and instructions share the same memory space. This architecture is also known as a Von Neuman architecture.

Program Counter (PC): The program counter is a simple counter generally found in a computer's control unit and whose output is generally used as an address that points to the next instruction in program memory to be executed by the program. The PC is typically expected to do standard counter microoperations such as parallel load and increment.

Program Flow Control Instructions: Instructions that cause or potentially cause the CPU to execute an instruction other than the instruction following the current instruction. Examples of program flow control instructions are conditional/unconditional branches, and subroutine calls/returns.

Program Flow Control: For computer programs to do useful things, they must appropriately respond accordingly to important "events". This response at a low level includes executing different portions of the given computer program. Computer instructions that facilitate any computer operation other than simple incremental execution of instructions from the program memory are generally referred to as program flow control instructions. Program flow control is generally handled by clever manipulations of the program counter.

Programmable Logic Device (PLD): Any integrated circuit used to create circuits in which the functionality of the internal circuit is not defined until the device is programmed (in this context, the term "program" does not typically refer to a computer programming language). One common type of PLD is the FPGA.

Programmed I/O: One of two main forms of computer I/O. Programmed I/O is characterized by dedicated instructions in the instruction set for performing data input and output. Programmed I/O synchronous in nature as it is associated with an executed instruction.

Programming Language Levels: Computer programs can be written on one of three general levels (listed from low to high): machine code level, assembly code level, or higher-level. Higher-level languages include C, C++, C#, Java, Wanker, etc.

Programming Model: The programming model, or *programmer's model*, describes the hardware resources available on a programmable computer-type device that the programmer is able to control via the program control. Program control is provided by the operations described by the device's instruction set and can either categorized as software or firmware.

Push: An operation associated with stacks where data is placed onto a stack; the stack pointer is appropriately adjusted.

-Q -

-R-

RAM: The acronym officially stands for Random Access Memory; a solid definition for RAM is fleeting due to advances in technology. RAMs are most often characterized as volatile, random access storage devices.

Random Access: A memory device is considered random access if it can access any of its contents in a constant amount of time. Devices such as flash drives are considered random access while devices such as tape drives and hard drives are not random access.

Read Access Time: The amount of time required for memory output data to become available after an address and the correct control signals have been provided to the device.

Register File: An abstract device that is used to model a given number of general purpose registers that are directly accessible by the given computers instruction set. Register files are typically modeled as multiport RAMs that can read and/or write multiple registers, roughly speaking, in a simultaneous manner.

Register Transfer Language (RTL): A syntactically loose approach to specifying a digital circuit that can be modeled as the synchronous transfer of data between sequential circuits such as registers. A RTL statement generally describes a microoperation (or set of micro-operations) generally associated with a digital circuit. The two parts of an RTL statement are 1) the register transfer specification, and 2) the specific conditions that are necessary for that transfer to occur. Generally speaking, only signals necessary for the stated transfer to occur are listed in the RTL statement while non-listed signals are assumed to be "properly handled" elsewhere. Unless explicitly stated, each RTL statement is assumed to occur in one clock cycle though the clock signal is rarely listed as part of the RTL statement. RTL is also known as *register transfer notation* (RTN).

Register: An n-bit wide sequential circuit that is primarily known for its ability to store bits. Registers are generally modeled as "n" D flip-flops which share a common clock. Register generally have synchronous parallel load inputs and sometimes other features (elementary operations) such as asynchronous or synchronous presets and clears. Specialized registers include shift registers and counters.

Retinal Persistence: The notion associated with the human visual system that does not allow humans to perceive an off-state of an LED at the exact time the LED is turned off. The notion of retinal persistence is what allows display multiplexing to work for humans.

RISC vs. CISC: The age-old computer argument of which is better that has never been solved. Generally speaking, RICS architectures require more instructions to complete a given operation than a CISC architecture would for that same operation, but those instructions are executed "more quickly" than a CISC architecture.

RISC: This acronym officially stands for "Reduced Instruction Set Architecture" and is generally used to describe computer architectures. In actuality, the term has little or nothing to do with the size of the instruction set. RISC architectures generally have the following characteristics:

- The architecture contains a register file with many general purpose registers
- The instructions word formats all contain the same number of bits (no extended opcodes)
- The instructions are executed in the same number of clock cycles
- The instructions generally are not overly complicated (meaning they don't generate great amounts of processing within the architecture)
- They have higher system clock frequencies than non-RISC architectures

ROM: The acronym officially stands for Read Only Memory; a solid definition for ROM is fleeting due to advances in technology. ROMs are most often characterized as non-volatile, random access storage devices.

-S-

Self-serving: The defining characteristic of all academic administrators and most engineering faculty.

Set-up & Hold Times: Digital devices that are edge sensitive (circuit changes state on a rising or falling clock edge) must hold inputs stable (the inputs must not change state) for a certain amount of time before the active clock edge arrives; this time is referred to as the *set-up* time. Digital devices must also hold the inputs stable for a certain amount of time after the active clock edge which is referred to as the *hold time*. Failing to meet set-up and/or hold times leads to the circuit going *metastable*.

Shadow Registers: A term used to describe storage elements for the C and Z flags as part of the RAT MCU context storage mechanism.

Shift Register: A special flavor of register designed to perform contiguous bit-level transfers (or serial transfers) of data between the bit storage elements of the register. Shift registers generally shift all the storage elements to a contiguous storage element once per clock cycle.

Software: In the specific case, software is a computer program that is written in a generic way so that it can run on a more than one type computer. Software does not refer to the language-level in which the program is written and thus can be written in machine code, assembly code, or a higher-level language. In the less specific case, the term software is often means any code written to run on a computer.

Spaghetti code: Programming code that does not follow standard structured programming concepts. Spaghetti code is by definition fragile; it is hard to understand, maintain, modify, and reuse.

Stack pointer: A term that refers to an entity that contains information that describes the "top of the stack".

Stack: An abstract data type that implement a last-in/first-out (LIFO) queue (or list of things). Stacks can be implemented in hardware or software with hardware implementation of stacks employing the use of a stack pointer to increase efficiency of the device. Stacks are typically used in computer architectures to keep track of hierarchically-nested processes such as subroutines and interrupts.

Standard Decoder: A standard decoder is a hardware device that implements a *one-hot* or *one-cold* output based on a given set of inputs. There is typically a binary relationship between the number of select inputs and the number of outputs and come in such flavors as 1:2, 2:4, 3:8, etc.

Start-up code: The code that is inserted automatically by the assembler as a result of declaring data in the program that requires initialization. The start-up code is typically comprised of instructions that initialize data memory.

Structured Code: Code that can be decomposed into three basic structure: 1) sequence, 2) if-then-else, and, 3) iterative. Structured code is easily understood, maintained, modified, and reused.

Subroutine: A set of instructions that a computer explicitly transfers to and returns from. In terms of program flow, the program transfers program execution to a set of instructions referred to as the subroutine. When the instructions in the subroutine have completed executing, control is returned to the instruction after the instruction, which caused the program to initially transfer to the subroutine.

Switch Bounce: A condition associated with all mechanical switches were upon actuation, the switch contacts make and break connections several times before the "settling" to the connected state. Switch bounce can last up to 20ms, depending on what source you consult.

-T-

Throughput: The throughput of a system is the total amount of useful information processed or communicated during a specified time period. Note that this definition is really general. Systems with high throughput are generally desired over systems with low throughput with the exception of administrative systems on university campuses.

Top-of-stack: A term that generally refers to the more recent item placed onto a stack.

Tri-State: A term that refers to a devices ability to effectively remove itself from a circuit. Thus a tri-state device in a digital circuit can either be high, low, or high-impedance. The notion of tri-stating is used to share routing resources in a circuit; the only possible drawback of tri-stating is that only one device can drive the resource at a given time, otherwise the condition of contention will occur, which is ungood.

-U-

Universal Shift Register: A special flavor of shift register that performs actions other than simple one-directional shifts including some or all of the following operations: shift left, shift right, barrel shifts, arithmetic shift, and rotates.

-V-

VHDL (Very High Speed Circuit Hardware Description Language): VHDL is one of several modeling systems referred to as "hardware description languages", or HDLs. VHDL is typically used to model digital circuits; the resultant models can be used to simulate circuits, or synthesize circuit implementations on PLDs or silicone.

Volatile/Non-Volatile: A device is considered volatile if its contents are lost when power is removed from the device while non-volatile devices retain their memory when power is removed and subsequently returned. The term volatile is most often associated with memory devices and PLDs such as FPGAs.

Von Neuman Architecture: A computer architecture where data and instructions share the same memory space. The term Von Neuman machine is often used to mean Von Neuman architecture. Von Neuman architecture is sometimes referred to as a "Princeton" architecture.

-W-

Write Cycle Timing: The amount of time required for data to be written to memory after a valid address, valid input data, and the appropriate control signals have been provided to the device.

-X-

X: The symbol typically used to represent input variables in finite state machines.

-Y-

Y: The symbol typically used to represent state variables in finite state machines.

-Z

Z: The symbol typically used to represent high impedance. This symbol is also used to represent output variables state machines.

INDEX

.

“

6

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z