John Seiffertt

Digital Logic for Computing

Digital Logic for Computing

John Seiffertt

Digital Logic for Computing

Springer

John Seiffertt
Truman University
Kirksville, Missouri
USA

ISBN 978-3-319-56837-9 ISBN 978-3-319-56839-3 (eBook)
DOI 10.1007/978-3-319-56839-3

Library of Congress Control Number: 2017937652

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

A traditional Introduction to Digital Logic course can cover all of Chaps. 1, 2, 3, and 4, selections from Chap. 5, and then all of Chaps. 6, 7, 8, 11, 12, 13, 14, and 17, and maybe selections from Chap. 19. I happen to think Chaps. 15 and 18 are of vital importance, and the case study presented in Chap. 20 really ties the whole book together, but they are not included in every intro course. What this book adds that others in the field leave out, through Chaps. 9, 10, and 16, is more depth for applications in the computing domain. They can be incorporated in an intro course geared towards computer engineers or computer scientists, and I have used this material myself, but they can also benefit these students simply by being in the text as breadth and a good reference for later use even if not directly included in the course.

The following descriptions detail the contents of each chapter.

Chapter 1—The Digital Electronic Computer

The goals of the text are introduced and the words *digital, electronic,* and *computer* are explored in relation to these goals. The block diagram for a computer is discussed, the digital abstraction is laid out, and the idea of programmability is discussed as core to our notion of a computer. The algorithmic approach of the text—the idea that this text is looking at the digital logic elements as the physical means by which algorithms are realized—is made clear. This chapter is short and not too technical and is recommended for all readers.

Chapter 2—Boolean Algebra

The heart of all mathematical engagement with digital logic is through the structure known as Boolean Algebra. We give a practitioner's appreciation for the ideas of Boolean Algebra while remaining rigorous and pointing to where the mathematics and philosophy underlying it all can take the interested reader. Often texts get bogged down in "axioms" and the like, but we focus on the uses of the tools of Boolean Algebra without watering down the sophistication. This is an absolutely core chapter as it deals with the fundamental definitions used throughout the text and the entire field of digital design.

Chapter 3—Logic Function Synthesis

Truth tables, canonical forms, and the first real stab at digital design properly understood are introduced in this chapter. Everyone reading this book will want to be quite familiar with this material. Many examples are walked through, and, combined with the analytic approach from Chap. 2, after finishing this chapter, the student will have a full grasp of what it means to specify logic functions and will have completed step 1 of a larger journey towards full understanding of what it means to implement algorithms in hardware.

Chapter 4—Basic Logic Function Minimization

Once we have logic functions generated with the methods of Chaps. 2 and 3, we often want to write them in more efficient forms so that our devices can cost less and consume less power. This chapter shows how to use Boolean Algebra to reduce logic functions to minimal forms and also introduces the Karnaugh Map. While more of a ladder idea than something of direct use to practicing digital designers, K-maps are ubiquitous in the literature of the field and the rest of this book uses them extensively to showcase minimal logic forms. It is worth it for the student of digital logic to at least be able to read a K-map and know what they are all about.

Chapter 5—Advanced Logic Function Minimization

The most important concept in this chapter used throughout the rest of this book is that of variable-entry K-maps. Almost every K-map presented in later chapters uses the variable-entry approach, so the reader is encouraged to read enough of this chapter to understand what they are all about. Working them requires more skill than the normal K-maps, so any particular reader may be forgiven for not wanting to pour too much energy into attaining complete mastery of the variable-entry maps. However, reading them and understanding how they let us get minimal forms is key. The other topics in this chapter are self-contained and can be skipped without loss of continuity.

Chapter 6—Logic Gates

This chapter introduces the schematics for the various logic gates for implementing Boolean Algebra operations. It also shows how to realize them in hardware using CMOS transistor arrays. While being able to read gate-level diagrams is of the utmost importance both for the digital design professional and for the student of the field, the transistor-level details are best suited for those going on to VLSI design work. The remainder of the text focuses on the gate-level and device-level rather than transistors.

Chapter 7—Unsigned Arithmetic

Working with numbers is essential in digital design and computing work. In this chapter, the student will learn about binary representations and how to construct addition and other basic arithmetic circuits from logic gates. Beyond the use of binary numbers, the material in this chapter is relatively self-contained. This is not a dedicated computer arithmetic text, and as long as the reader understands the device block diagram for them, the remainder of the chapters ought to be accessible.

Chapter 8—Signed Numbers

The important two's complement representation is covered. It cannot be overstated how important this is to readers of this work because of the ubiquity of two's complement in all manner of digital devices and computers. Other representations are also covered, as are signed arithmetic devices.

Chapter 9—Other Digital Representations

Not everything we need to encode within a digital system is a number. This chapter discusses how to design such representations and emphasizes that overall the way we express any information is, in fact, an actual design decision. Floating point numbers are discussed as well, showing how non-place-value representations composed of fields can be used to form useful encodings. While the BCD representation is referenced throughout the rest of the text, much of this chapter is self-contained, and the reader who only wants to study 19 chapters could probably hold off on this one with minimal effect on the rest of the material.

Chapter 10—Encoding Code

This chapter highlights the text's unique focus on computing by extending the work of Chap. 9 in showing how to represent computer programs themselves as digital information. Assembly language concepts are introduced, and we begin to see the development of datapaths to support them. How we write selection and control programming constructs at the machine level is discussed, and for technical readers who have never before seen this material it is quite the ride. This is a highly recommended chapter, especially for computer scientists and computer engineers. One of this book's unique features is that it ties in later as we often use these instructions as motivating design elements for our datapaths and datapath controllers.

Chapter 11—Sequential Logic Elements

Sequential logic is a vital, if not defining, feature of digital design. It is through sequential circuits that we can store information, implement algorithms, and really harness the power of our digital systems to create tools of such awesomeness that integrated circuitry is arguably the most transformative technology man has ever seen. This chapter discusses flip-flops, synchronization signals, and timing diagrams.

Chapter 12—Multiplexers and Comparators

This chapter begins our march towards the full development of digital datapaths and the implementation of algorithms in hardware. The two devices covered here, multiplexers and comparators, are shown in the context of the various types of instructions they can help realize as well as their use in algorithms. These devices really are core to the field. Every reader of the text should know about them.

Chapter 13—Decoders and Register Files

Building on the development of Chap. 12, this chapter introduces more important datapath components and shows how they are used in designs. The decoder, in particular, is used to illustrate how logic functions may be implemented and gives the student a strong workout in Boolean Algebra and active-high vs. active-low logic.

Chapter 14—Counters

The state machine design process, core to the use of sequential networks, is introduced in this chapter in the context of building counters. Many kinds of counters are presented and their use in datapath design and signal generation is discussed. As counters are of critical importance in digital design, and the state machine design process is good to learn in its simplified form presented here first, this is a highly recommended chapter.

Chapter 15—Datapaths

This is an important design chapter. The study of datapaths brings together all the components from previous chapters and ties them into the algorithm concepts at the core of the text. A successful study of this chapter will help the reader to learn to take specifications, optimally lay out the design using digital elements, and minimize the required signals for maximum efficiency.

Chapter 16—Basic Computer Datapath

A combination of sorts of Chaps. 9 and 15, in this chapter we discuss how to construct a datapath of particular importance: that of an actual digital electronic computer. Of definite interest to computer scientists and computer engineers, even electrical engineers may enjoy seeing this key specialized application of digital circuitry.

Chapter 17—State Machines

This is one of the most important chapters. No study of digital design is complete without full coverage of the design and application of state machines. We begin with a generalized state machine partitioned into next state logic, output logic, and state memory. This chapter then develops state diagram, state tables, and implementations using various flip-flops. Mealy and Moore machines are covered.

Chapter 18—Datapath Controllers

Extending the initial discussion of state machines from Chap. 17, this chapter looks at their use specifically in conjunction with the datapaths covered in Chap. 15. Partitioning large digital systems into state machine controller and datapath sections is common, and through detailed examples this chapter walks through the lengthy design process. The climax of the text in a way, the design of datapath controllers brings together everything previously studied. The student who successfully works through this chapter has obtained a strong intro-level knowledge of digital design and is well-positioned for more advanced study in the field.

Chapter 19—State Machine Theory and Optimization

State reduction and assignment are covered in this chapter, and those are reasonably standard in texts of this kind. This chapter begins, however, with a discussion of formal languages and automata. These are topics typically reserved for computer science theory courses, and they are included here to emphasize the text's unique focus on computing. The relationship between theoretical models of computation and their applied counterparts the student has been construction in previous chapters is made clear. Not intended even as a primer in automata theory,

the goal here is to give just a brief morsel to entice further thinking on the matter. Too often, the science and engineering sides of the field are kept very far apart from one another, and this chapter, by drawing parallels between them based on material relevant to both and covered previously in the text, is an attempt to bridge this gap. It is certainly not going to serve as a compete study of these topics, but the hope is that some philosophically inclined readers may see these ideas here for the first time and be motivated to explore them further.

Chapter 20—Instruction Processor Design

The text ends with a full-design case study of an electronic digital computer. We take a pedagogical approach here in repeating some concepts from earlier chapters, intentionally using slightly different language and notation, so the student can see them really coming together in a solid whole. The idea is that if you try to read this chapter first, it should feel overwhelming. But, if read after careful study of the rest of the text, the successful student should be nodding along and say "Yes, I have this!" as the actual computer is designed from the instruction set on up. The exercises continue the design process and, if completed, the reader will have finished the study of digital logic for computing by actually sketching the details of a functional instruction processor. We feel this chapter is a unique feature of this text and encourage students to take advantage of it.

Acknowledgments

This work is the result of more than 5 years of teaching digital logic at the Missouri University of Science and Technology. I thank most of all the fine students of that great institution whose questions, comments, and feedback shaped my course and formed the bulk of this text. I am also grateful for the support provided by my colleagues, in particular Joe Stanley and Donald Wunsch, who taught alongside me and motivated me to increase my teaching ability both in the classroom and in the preparation of this manuscript. Many excellent problem types and modes of explanation found in this text are due to their insights. I also want to thank my current department at Truman State University for their encouragement as I completed this text.

Contents

Chapter 1
The Digital Electronic Computer

Welcome to the study of Computing!

Computers are the transformative technology of our times. In addition to powering our phones, music players, and laptops, they control the operation of our cars, maintain the safety and efficiency of power plants, enable low-cost medical equipment to save lives in remote locations, power GPS devices, and run nearly all home appliances and electronic toys of every type.

This book will introduce the core concepts and methods associated with the design and physical construction of digital electronic computers.

Working through this book you will learn how to build a digital electronic computer from the ground up. This includes understanding the digital abstraction, seeing how we use electronics to implement our digital system, what digital circuit elements are important for computers, learning how we relate our high-level programming constructs and needs to the low systems level, and then bringing it all together with an overview of basic computer organization.

Specific topics include number systems, Boolean algebra, truth tables, computer arithmetic, logic gates, CMOS, flip-flops, instruction sets, adders, decoders, multiplexers, comparators, finite state machines, and datapath/controller systems.

Those with interest in electrical engineering can use this course of study as a starting point for investigations into digital electronics while computer scientists will see how their use of computers is affected by the underlying design of the machines. Computer engineers will appreciate the entirety of the book as it serves as an overall introduction to their field in all its diversity.

We conceive of a computer as seen in Fig. 1.1.

We first design the digital logic elements, from the transistor level on up, to build the datapath component of the computer. It is in this component where the core "computation" activity that we most associated with instruction processors occurs. The control unit is heavily discussed in the context of state machines as an algorithm level topic. Later chapters in the text walk through the construction of this important component. The memory unit is covered only indirectly; it's much more of an intense electrical engineering topic to get into the deep details of how

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_1

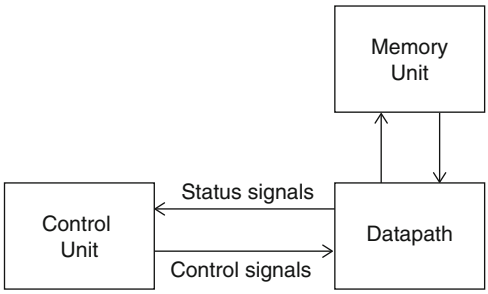


Fig. 1.1 Block diagram for computer system

computer memory is constructed today. It suffices for our discussion to think of it as a store of values accessed via labels called addresses. It is from that context that we operate when viewing the entire computer block diagram as a functioning whole. Advanced courses on computer architecture and organization will get into the details of how the interface between memory and the CPU proper (understood to be the control unit-datapath blocks taken together) is handled. Discussions of cache memory and its associated tradeoffs and optimizations lay beyond the scope of the current work.

While the methods and tools of digital logic design are widely applied to many important electronics applications, and many of these are touched upon herein, we do use the computer as the overarching motivating example in this text.

Fundamental to our approach is looking at everything from the perspective of the abstract realities known as *algorithms*. At the heart of computing science is found the study of the properties of algorithms, and it can be argued the goal of computer engineering is the deployment and application of these *algorithms*. Therefore, we look at the development of digital logic circuits as the hardware implementation of algorithms. Our approach to motivating individual components such as multiplexers and decoders is through the lense of how they contribute to the overall workings of the algorithms. Hence, the title of the book: Digital Logic for Computing. We're focusing on taking the algorithms from the world of code to the world of metal.

Digital

We can physically compute with analog values in a number of ways. We can pour water from two buckets into a third: the water now present represents the sum of the previous two values. We can attach two wires together and see the total voltage being the sum of the voltages of each. While it may seem ideal to compute in this fashion because we can represent any number and perform calculations to arbitrary precisions, what happens in practice is that we lose tiny bits here and there. A few drops of water fall from a bucket and the wire degrades a bit of the voltage.

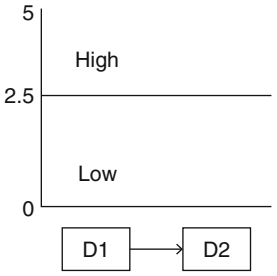


Fig. 1.2 High-low digitized transmission

All together, this interference, this **noise**, makes precise computation difficult. It is for this reason that we consider **digital** systems in this book.

The word *digital* derives from the Latin *digitum*, which means *finger*. It was originally used to mean ten: this is why we have *digits* in our base-10 number system. In fact, early precursors to modern computers such as Charles Baggage's Difference Engine, were digital in this sense: they operated mechanically using base-10 numbers. We're going to broaden the sense of the word and refer to a **digital system** as one whose values are *digitized*, that is, whose values are broken into discrete chunks rather than being capable of representing every possible number.

Due to the limitations of our underlying technologies, we're going to digitize to the maximum extent: we will use only two values within a given range of possible values. Consider the graph in Fig. 1.2.

We divide our infinity of values from 0 to 5 into two and we call them High and Low. Device D_1 is transmitting a signal to device D_2. In order to communicate effectively, D1 must be assured that when it sends a signal that D2 will be able to properly interpret it as either High or Low. We see that this system has some resistance to noise: a signal of 4.5, for example, subject to a shock of .75 might end up transmitting as 3.75. But that is still OK because it's in the High range and D2 still interprets it as a High signal. If we were pouring 4.5 ounces of water from one cup to another and lost .75 ounces along the way we'd have a wildly inaccurate sum. But in a digital system we're fine and the computation is not affected by the noise.

But, what if we transmit a High signal that's less strong, say 3.0? It's still within the High range so our device D1 is satisfied. What if it is hit with a disruption of .75? Now it's transmitting a 2.25 signal which D2 interprets as being in the Low range. We now have a breakdown in communication and our digital system won't function. To fix this we need to force the High signals to be really high and the low signals to be really low. We need to ensure separation between our two values.

Consider the graph shown in Fig. 1.3. Now instead of breaking the entire range of values 0 through 5 into two categories, we have that High signals are in the range VH to 5 and low signals in the range 0 to VL while any signal in the range VL to VH is in the forbidden zone and is therefore not interpreted. Provided we can get VH close to 5 and VL close to 0, we've done a good job here of not confusing High and Low signals. If we have our 3.0 signal with the .75 shock, so long as $2.25 >$ VL we

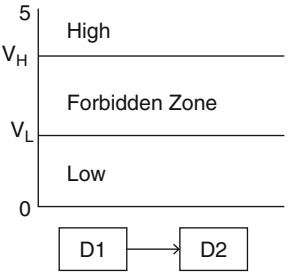


Fig. 1.3 Using the forbidden zone to help digitize transmissions

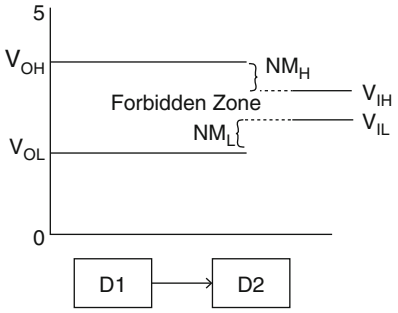


Fig. 1.4 Noise margins for digitized transmissions

now can be confident that D2 will not misinterpret our noisy transmission. Instead, D2 picks up a signal in the forbidden zone. While that's not ideal, it's a better case result than D2 not even being aware a problem exists.

But, we're not done! We can improve yet upon this configuration. We still have an issue if we're sending a signal that is close to VH or VL because just a small amount of noise can disrupt it and send it into the forbidden zone. What we'd like is for D2 to recognize signals close to these breakpoints that fall just a little bit into the forbidden zone. This is demonstrated in Fig. 1.4.

If we make the forbidden zone smaller for the **input** values then the receiving device can better understand what the transmitting device is sending even through a bit of noise. We can now talk about VOH and VOL as the output high and low value cutoffs and VIH and VIL as the input high and low value cutoffs. The high and low noise margins NMH and NML can be defined:

$$NM_H = V_{OH} - V_{IH}$$
$$NM_L = V_{IL} - V_{OL}$$

It's not quite as simple to digitize our systems as we may think at first. Just because the devices "are all 1's and 0's" doesn't mean their internals are easy to design. There are a lot of moving pieces here.

We speak of **Logic Families** as being devices that can work together. That is, digital devices within a logic family have their high and low value cutoffs aligned so that they are always assured accurate transmission. We can also look into

Table 1.1 TTL and SMOS logic values

	VIL	VIH	VOL	VOH
TTL	0.80	2.00	0.40	2.40
CMOS	1.35	3.15	0.33	3.84

intra-family communication. Consider Table 1.1 which gives the details for two logic families, TTL and CMOS.

We can ask whether a TTL device can transmit to a CMOS device? The VOL of TTL is .40 which is less than the VIL of CMOS at 1.15 Therefore, we have a NML in this situation of 1.15–0.40 = 0.85. Looking at the high side, however, we see that TTL's VOH is 2.40 while CMOS's VIH is 3.15. This means the TTL device can transmit any value greater than 2.40, say 2.75, while the CMOS device won't recognize any input less than 3.15 as a high signal. Therefore, communication fails in this case. A TTL device cannot transmit reliably to a CMOS device.

What about the opposite case? Can a CMOS device transmit to TTL? Well, on the Low side the CMOS device has a VOL of 0.33 which is less than TTL's VIL of 0.80 for a NML 0f $0.80-0.33 = 0.47$ which is positive and therefore compatible. On the High side, we have CMOS with a VOH of 3.84 and TTL with a VIH of 2.00. This gives us a NMH of $3.84-2.00 = 1.84$ which is fine. Therefore, we can indeed have a CMOS device transmit to a TTL device.

Be sure to carefully go through these computations so you have the idea of noise margins, High, and Low signals clear. The entirety of digital design rests upon this foundation, so it's a good idea to get to where it makes sense to you.

We want our digital devices to be able to effectively communicate. That is, when a device sends a logic-1 (high voltage) we want the receiving device to read a logic-1 and when a device sends a logic-0 (low voltage) we want the receiving device to read a logic-0.

To this end, devices have tolerances for the voltage levels they interpret as logic-1 and logic-0 when both sending and receiving signals. In order to build devices that are as resistant to noise as possible, we have different tolerances for output and input signals. The values V_{OH} and V_{OL} tell us the cutoff voltages for high and low output and the values V_{IH} and V_{IL} tell us the cutoff voltages for high and low inputs.

A device can drive another if its V_{OH} is less than the V_{IH} of the receiving device and if its V_{OH} is less than the V_{IH} of the receiver.

A device can receive signals from another if its V_{IH} is less than the V_{OH} of the driver and if its V_{IL} is greater than the V_{OL} of the driver.

If devices of two families can both drive and receive signals from each other, then the families are said to be compatible.

Electronic

We can implement digital systems using a variety of underlying technologies. The basic electronic element we're going to use to design our systems is the **transistor,** a device that will either block or permit a signal to pass. We'll interpret blocking the

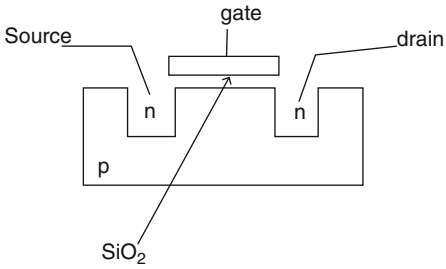


Fig. 1.5 A MOSFET

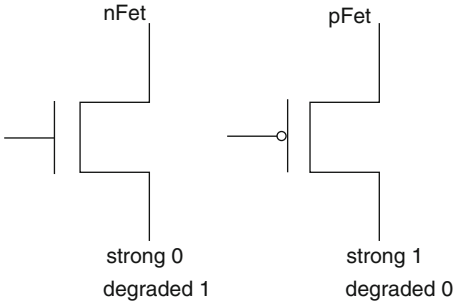


Fig. 1.6 nFets and pFets

signal as our Low digital value and allowing it to pass as our High digital value. In particular, the **MOSFET: metal oxide semiconductor field effect transistor.** In Fig. 1.5 we see a diagram outlining its construction.

The electric signal moves from the **source** side out the **drain** side if the voltage on the **gate** is high enough. Chemically, to effect this transition we use two materials, called **n** and **p**, each doped with specific metals (n material is doped with group II metals and p with group III metals.) We end up with two kinds of MOSFETS, depending on where we use the n and p materials. In the diagram, we have an **nFET** device. If we swap the n and p materials, we would have a **pFET** device.

It turns out that nFET and pFET devices have different characteristics. Transistors in the nFET category transmit a Low signal very well but have a hard time passing a High signal. They end up pulling the High voltage lower, and sometimes this is more noise than the device can handle and we end up in the forbidden zone. The pFET devices are the opposite: they are just fine for passing High voltage signals but pull Low voltages up and make them harder to read.

In technical language, we say the nFET passes a **degraded 1** and a **strong 0** while the pFET passes a **degraded 0** and a **strong 1.** (In later chapter's we'll call the High voltage a logic-1 and the Low voltage a logic-0.)

The diagram in Fig. 1.6 shows the circuit symbol we use for nFETs and pFETS as well as summarizes the characteristics relevant for our digital design needs.

In Chap. 6, we'll get into great detail about how we use these transistors to build the components necessary to implementing algorithms with our digital logic

elements. For this introductory chapter, it's sufficient to leave off here with the idea of underlying technology merely described at a low level.

Computer

In this book we will study the design, operation, and capabilities of the digital electronic computer, the most widely used commercial embodiment of the idea of an instruction processor.

But what do we mean by the word *computer*? Students on the first day of class often respond to the question "What is a computer?" with a variation on "something that computes!" The natural follow-up is then "ok, so what is a computation?" That one is harder to answer. It's common for non-CS people to think of computation as "coming up with fancy ways to calculate the square root of a number" and other related numerical investigations. And while we certainly want to include the exploration of number within our understanding of computation, we don't want to commit to that being the end of the story. Because, more generally, and as stated in the beginnings of the field of computing science by Alan Turing and other pioneers, when we think of computations in a true computer science sense we mean not the processing of numbers only but the processing of language taken more generally.

That's right! Computations are considered the processing of language! And, this dovetails nicely with the approach taken in looking at digital logic from the vantage point of *algorithms* broadly understood. Because, at the end of the day, some of the most important things we can say about algorithms involve their expressibilty as language. The code we write to configure instruction processors to do great things is, after all, language, and we consider the computer itself as a digital electronic circuit that responds to this language.

So, the language, more specifically the idea of *programmability* which involves the processing of not just one sequence from a language but rather the ability to handle any expression of a particular language, is central to our understanding of what a computer is. While we may take ancient devices such as the astrolabe or abacus as computer in the sense that they do indeed take inputs and produce outputs based on some internal mechanized computation, they are not programmable so they do not qualify as a computer.

We can go a step further and look at digital electronic circuits, such as those in a simple hand calculator, that may excel at computations and, as we'll study as we proceed in this text, are indeed the implementation of algorithms expressed in language. But, these are also not computers because they are not programmable! You cannot change what they do by writing a new algorithm. These are called application specific integrated circuits (ASICS) and are the focus of electrical engineers when they study digital design principles. We'll certainly talk about them, and Chap. 18 in particular develops these in great detail, but the text caps with a chapter on the design of an instruction processor and throughout we are concerned with the relationship between digital logic and programmability.

Exercises

1.1 Research the Antikethera mechanism. It is sometimes referred to as the first computer. It's certainly not electronic, but is it digital? Also, does it satisfy our understanding of *computer* as described in this chapter and pursued in this text?

1.2 In this course we focus on digital electronic computers. However, it is possible to have computers that are neither digital nor electronic. For this problem, research historical computers from the nineteenth or twentieth century that are analog and mechanical. Write a few sentences about each and indicate the source of your information.

1.3 Consider the following logic families:

	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5	.8	2.0	.4	2.4
CMOS	5	1.35	3.15	.33	3.84
LVTTL	3.3	.8	2.0	.4	2.4
LVCMOS	3.3	.9	1.8	.36	2.7

Answer each of the following questions YES or NO and, if NO, give a reason.

(a) Can a TTL device drive a CMOS device?
(b) Can a TTL device drive a LVTTL device?
(c) Can a TTL device drive a LVCMOS device?
(d) Can a CMOS device drive a TTL device?
(e) Can a CMOS device drive a LVTTL device?
(f) Can a CMOS device drive a LVCMOS device?
(g) Can a LVTTL device drive a TTL device?
(h) Can a LVTTL device drive a CMOS device?
(i) Can a LVTTL device drive a LVCMOS device?
(j) Can a LVCMOS device drive a TTL device?
(k) Can a LVCMOS device drive a CMOS device?
(l) Can a LVCMOS device drive a LVTTL device?
(m) What does "LV" stand for in the LVTTL and LVCMOS families?

1.4 Consider the following logic families:

Family	V_{IL}	V_{IH}	V_{OL}	V_{OH}
A	1.0	4.0	0.5	4.5
B	2.0	4.0	1.5	3.5
C	0.5	3.5	0.25	4.0
D	1.5	4.5	1.25	4.75

Which combinations of families are compatible?

1.5 Consider the following logic families:

Family	V_{IL}	V_{IH}	V_{OL}	V_{OH}
A	1.0	4.0	0.5	4.5
B	2.0	4.0	1.5	3.5

(a) Can a device from family A drive a device from family B? Why or why not?
(b) Can a device from family B drive a device from family A? Why or why not?

1.6 Suppose a logic family has a high noise margin of $NM_H = .3$. If $V_{IH} = 4.5$, what must V_{OH} be?

1.7 There are many ways to design a device which will sequentially execute instructions. The mathematician John von Neumann was a pioneer in the development of digital electronic computers, consulting on the construction of a computer called the EDVAC in the 1940's. His early writings on computer design and organization are still relevant today, as most computers are built using the basic principles he laid out. Research the concept of *von Neumann architecture* and write a few sentences about its features.

1.8 Is a sun dial a computer? It takes an input, sunlight, and computes an output, time of day? So, is it a computer? This is a bit of a jokey question, but also highlights the importance of the key elements we want in our definition of a computer. We're not particularly interested in exploring what a pole stuck in the ground can do.

Chapter 2
Boolean Algebra

We need a language to discuss the design of the computer. Once upon a time, humans looked to the stars and developed what we know as Calculus to describe their motion. It turned out that algebra and calculus based on the real numbers work quite well to describe systems involving energy, motion, time, and space. The real numbers are a useful model for those quantities. Within our digital computer, however, we only have access to two values, logic-1 and logic-0. We don't have an infinity of values over which our functions may vary. Therefore, the early pioneers of computing found themselves faced with a dilemma. They could not use the classical mathematics they knew and loved; rather, they faced the task of developing a new mathematics capable of working with only two values.

These engineers were not amused. Developing new mathematics is for mathematicians, they reasoned. So, they did the next best thing. They looked around at what the mathematicians were doing and instead of crafting a new computer design mathematics from scratch they happened upon something close to what they wanted, and they stole it. (OK, they repurposed it.)

It turned out that a nineteenth century schoolteacher and philosopher named George Boole had worked out a mathematics for how people (or at least how mathematicians) reasoned through formal problems. Instead of representing quantities such as velocity and mass and curvature, the variables in Boole's mathematics represented English statements. Instead of addition, subtraction, and multiplication, Boole's mathematics combined variables in the logical operations of and, or, and not. So far, this isn't really what the computer engineers were looking for, but the last key piece of what Boole had developed sealed the deal: all the variables took on only one of two values! That was it! The engineers needed a two-valued mathematics, and Boole had one. Sure, Boole called the values True and False and the engineers wanted to call them logic-1 and logic-0, but it was close enough (engineers are all about "close enough".)

So, here we are: Boolean Algebra is the name given to the mathematics that is used to describe the behavior of all digital systems, computers included. Let's check it out.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_2

Digital Logic Design Process

To get a sense of how Boolean Algebra differs from the algebra and calculus you're already familiar with, let's consider an example. Suppose we have a cabin by a river and staff it with a robotic sentry to keep out bears and ewoks. The house is equipped with window, door, and chimney sensors which detect intrusion as well as an infrared sensor which can determine size, and the sentry can sound a local alarm or take action against the intruder.

The schematic of the setup can be seen in Fig. 2.1.

We call the door sensor D, the chimney sensor C, the window sensors w_0, w_1, w_2, and w_3 (in the diagram w_3 is overlooking the river), and the infrared sensor we'll call R. Let's give names to the sentry's actions, too: A will be sound the alarm and E will be to engage the intruder.

Now, let's see how Boolean Algebra can help us specify how this system works. First, we would like the sentry to sound an alarm if any of the door or window sensors activate. We can write this as an equation:

$$A = D \text{ or } w_0 \text{ or } w_1 \text{ or } w_2 \text{ or } w_3$$

Yes, that is a Boolean Algebra equation because the logical or is considered a mathematical operator. The logicians who work with this use the symbol $\vee$ to indicate logical or and would write this as $A = D \vee w_0 \vee w_1 \vee w_2 \vee w_3$. While that's not so bad, we'd like to avoid new symbols all together if we can. So, in the Boolean

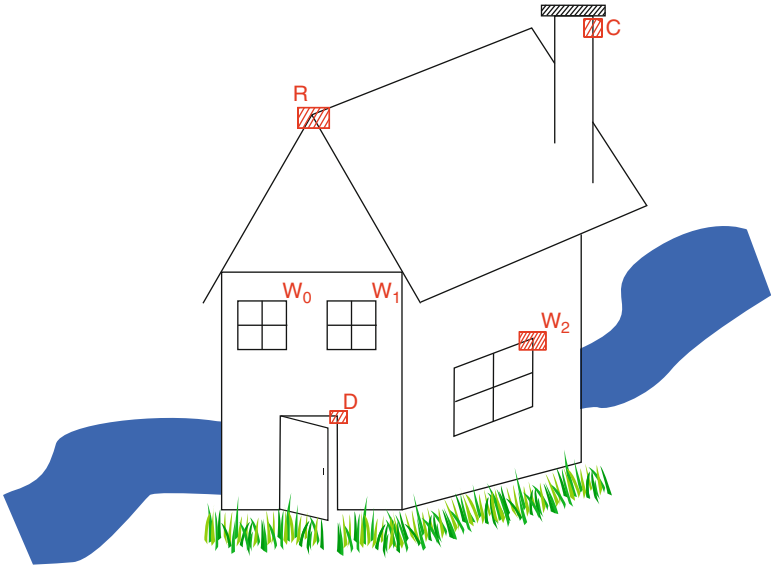
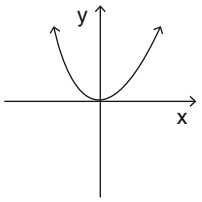


Fig. 2.1 Boolean Algebra Schematic: D is the door sensor, W0, W1, W2 are the window sensors, R is the infrared sensor, and C is the chimney sensor

Fig. 2.2 A parabola



Algebra digital designers use, we represent the logical or operation with a plus sign, +, and our equation becomes

$$A = D + w_0 + w_1 + w_2 + w_3$$

Now we're getting somewhere! We have an equation where the variables can take on two values, logic-1 and logic-0, that implements the behavior we've specified. But, we start to wonder, what does it mean to *add* these strange logic-1 and logic-0 values? I know what 15 + 6 means, but what does logic-1 + logic-0 mean? To answer that question we have to specify how our OR operation works.

In real-valued algebra and calculus we often graph functions. For example, $y = x^2$ gives us the familiar parabola (Fig. 2.2).

The arrows on the axes and the ends of the parabola indicate these quantities extend forever, as the real numbers are infinite. We also know the solid lines of the shape indicate an infinitely dense system of numbers, as no matter how deep we delve, between 2 and 3, or between 2.5 and 2.6, or between 2.55 and 2.56, there is always another number to be found.

In our two-valued system, this is not the case. We don't need to specify how our operators work using a graph as we do on the real line. Instead, we use a table that enumerates all possible combinations of inputs, because (at least for small numbers of input variables) this is quite feasible.

So, here's the table that tells us how our OR operation works:

On the left we have all four combinations of our two input variables and on the right we say what the OR operation returns for each combination. Taken as normal math, the first four make a lot of sense, as there is nothing objectionable to noting that 0 + 0 = 0, 0 + 1 = 1, or 1 + 0 = 1. However, the last line, claiming 1 + 1 = 1, gives us pause. It's clear now we are not working with real numbers. We are in another world, that of Boolean Algebra. We must remember that the 1's and 0's are not the same 1's and 0's that we find on the real number line and the + is not the same addition we apply to real numbers. Only the equal sign is the same! For us, 1 + 1 = 1 is natural and explains how we want our digital system to work: we want the entire expression to evaluate to logic-1 as long as any one of the inputs is logic-1. Going back to our example, we see that we want the alarm to sound, that is, we want A to be 1, when any of the sensors is 1, so it would be counterproductive to declare that 1 + 1 = 0 (which is the only other possibility since we don't have the symbol 2 available to us) because that would mean that intruders could break in if only they attack two locations simultaneously. That's not the behavior we're looking for here.

Table 2.1 is called a **truth table** and the values of the function are called **truth values**. These terms are artifacts from the original application of Boolean Algebra to matters of propositions, arguments, and the truth of statements. It's incredibly important that we realize that while we are using the same underlying mathematics, our interpretations of the two values in the system are fundamentally different from traditional Boolean Algebra. For us, the terms True and False **have no meaning**. We say "one" and "zero" instead, and, while there is some obvious connection symbolically between our logic-1 and True and between our logic-0 and False, carrying that connection too far can impede our ability to design systems effectively. We may think that logic-1, or True, means that a particular device is "on" or "active" and a logic-0, or False, means the device is "off" or "inactive." **Nothing could be farther from the truth!**

We've skipped over this detail so far, but it's actually crucial to decide what logic-1 and logic-0 represent in a system. If we say that the variable D = 1 when the door sensor detects movement and D = 0 when the door sensor does not detect movement, then we are saying the sensor is **active high**, that is, that the logic-1 corresponds to the device being on, or active. However, this doesn't have to be the case (and, depending on how the electronics are designed, is often not the case.) We could specify D = 1 when the door sensor does not detect movement and D = 0 when the sensor detects movement. Then D would be **active low**, that is, the logic-1 value would be associated with an inactive device and the logic-0 value with an active device (Table 2.2). If we think in terms of True and False these electronic realities can be hard to grasp. It's much better to think in terms of abstract 1's and 0's whose meanings are determined by the behavior of a given real physical system.

Let's look back at our example. We've assumed the signals A, D, w0, w1, w2, and w3 are all active-high. What happens if, say, the window sensors become active-low? Now we want the equation for A to evaluate to a 1 when we get a 1 on D or a 0 on any of the wi's. To account for this, we need to introduce a new logic operator: not.

Table 2.3 shows the truth table for not.

Table 2.1 Logical OR operation

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.2 Summary of active-high and active-low terminology

Active-high devices generate a logic-1 signal when active and a logic-0 signal when inactive
Active-low devices generate a logic-1 signal when inactive and a logic-0 signal when active

Table 2.3 Logical NOT operation

x	not-x
0	1
1	0

Once you grasp it, this is all pretty straightforward and simple, yet it remains incredibly useful. We can now keep the basic form of our logic equation for the alarm A and just replace the active-low signals with their complements (the not of a variable is called its *complement*.)

$$A = D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3}$$

If A were active-low, we'd need to complement the entire equation and end up with

$$A = \overline{D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3}}$$

The idea is that the sentry will sound the alarm to scare off weak or small intruders. What if we have a large bear smashing through the window? In this case we don't want to sound an alarm which may only serve to further enrage the beast. So we want our (active-high) Alarm not to sound when the infrared sensor detects a large creature. Let's say that happens when R = 1. So, in words we want A to be 1 when the door sensor is active or when any window sensors are active and when the infrared sensor is inactive.

We have a new logic word: and. Our equation becomes

$$A = (D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3}) \text{ and } \bar{R}$$

While the logicians would use л for *and* (and call it *conjunction* to boot), we're not going to adopt all their practices and we'll go ahead and use our familiar multiplication symbols for the logical and operation:

$$A = (D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3})\bar{R}$$

The truth table for and shows us that this operation is only logic-1 when all its inputs are logic-1 can be seen in Table 2.4.

The multiplication makes sense here as we know anything times zero is zero. It's intuitive and works, so we'll go with it.

What about the chimney sensor? The idea here is that any intruder bold enough to enter through the chimney needs to be met head on by the sentry, same as any intruder sized too large. Let's make the chimney sensor active-low so that C = 1 when no intruder is detected and C = 0 when one is scurrying down the smokestack. Our alarm equation is then

Table 2.4 Logical AND operation

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

$$A = (D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3})\bar{R}C$$

Read over this logic and make sure you follow why we have a C at the far right. It's easy to get mixed up when dealing with nots and active-low logic.

Let's write the equation now for the sentry's engage logic. We want it to engage when the intruder is large or activity is detected on the chimney sensor (that means R = 1 and C = 0).

$$E = R\bar{C}$$

We could add a timer T to the system that measures how long the alarm rings. If the alarm rings long enough to exceed a threshold (T = 1) then the alarm should shut off and the sentry ought to engage the intruder because the alarm clearly isn't working to scare it away. Then our final equations become

$$A = (D + \overline{x_0} + \overline{x_1} + \overline{x_2} + \overline{x_3})\bar{R}C\bar{T}$$
$$E = R\bar{C} + T$$

Make sure you see where these equations come from and you'll be on your way to writing logic equations to describe the behavior of digital systems.

The Mathematics of Boolean Algebra

What we've covered so far are the basic elements of logic operations that we'll need to use the rest of the book in designing of our digital systems. There is a lot more to Boolean Algebra and the mathematical approach to logic than we will actually require problem-to-problem as digital designers. This section delves deeper in the mathematical structure underlying Boolean Algebra and illustrates some more ways to conceive of and to work with the symbols we generate to describe our systems in terms of AND, OR, and NOT.

Mathematics is about getting to the essence of certain kinds of things and then drawing connections among these essences. Where laymen see a 5 and a plus sign + and a 7 and think "Yes, 5 apples and 7 apples makes a dozen, thank you very much," mathematicians ask things like "what, really, are 5 and 7," what is "addition," and "what fun is there to be had if we can tinker with each of these?"

The normal answer is that 5 and 7 are **real numbers** and addition + is an operation on these real numbers. But what if 5 and 7 were instead members of the set {1,2,3,4,5,6,7,8) and addition were now an operation on this set of integers? We'd have to define 5 + 7 in another way. Maybe we'd just wrap around so that 5 + 7 = 4. We can call this modular arithmetic and run with it a great distance, tinkering this way and that as our system gets more and more interesting.

Algebra's essence, then, is that we have some variables which take on values from a set (or sets), we have operations that follow certain rules, and we have some

special elements, 0 and 1, say, which have unique properties (such as anything added to 0 or multiplied by 1 gives the same thing back.)

That's it! We can talk about polynomial algebras in which the variables represent entire polynomials and the operations are addition and multiplication of polynomials instead of numbers. There are matrix algebras where we take matrices as the variables and (try to) add and multiply them and vector algebras where we let our variables range over not one but two different sets with all sorts of different rules for how the operations apply to variables that are now not even the same type of things. We can specify types of algebras even more precisely and talk about groups, rings, fields, and even universal algebras. It's all great fun and really beautiful and can reward a lifetime of study. We don't have a lifetime here, so we'll talk about our specific algebra: that of George Boole.

Now, Boole didn't write this all down the way modern mathematicians think of it. The twentieth century saw mathematics taking a quantum leap in generalization (I blame it on topology) and therefore we now have quite a few ways to talk about things like "rules" and "axioms" and "properties" of "Boolean Algebra," which is why you can read about them in an engineering text and a mathematician's tome and not even realize you're talking about the same thing.

For example, a computer engineer may say "Boolean Algebra is what I use to build my components out of logic gates because it's all about 1's and 0's" and then mutter something about 74-series chips as he reaches for his ruled pad while a mathematician may claim "Boolean Algebra is a complemented distributive lattice with at least two elements" and then mutter something about ultrafilters as he sharpens his pencil.

They are both right and they are both referring to the same beautiful construct. They are just using it and interacting with it in different ways. Since our goal is to build a computer, we're going to (mostly) approach it from the point of view of looking more deeply at how our basic logic operations interact with each other so we are more familiar with their characteristics. Our hope is that this will help us conceptualize our high level designs.

To that end, we're going to talk about two kinds of rules: those that make you go "duh" and those that detail the interaction among our operations.

In the first category we place things like "x + 1 = 1" and "x * 0 = 0". These are basic identities that, while fundamental, don't need a lot of motivation. They are collected in Fig. 2.3:

Most of these make intuitive sense. The fact that $x+1=1$ is due to the fact that we don't have a "2" symbol so we can't go higher than 1. Once we're at 1, we're as

Fig. 2.3 The "common sense" Boolean Algebra identities

$$x + x = x \qquad x + 1 = 1 \qquad x \cdot 0 = 0$$

$$x + \overline{x} = 1 \qquad x\,\overline{x} = 0 \qquad x\,x = x$$

$$x + 0 = x \qquad x \cdot 1 = x \qquad x\,y = y\,x$$

$$x + y = y + x$$

high as we can go. The OR operator, represented by the +, is a "1's detector" and once a single logic-1 is involved the entire operation is guaranteed to evaluate to it.

In the next category are important things called distributive properties and De Morgan's Laws which have purchase beyond the realm of computing.

The **distributive properties** tell us how our logical AND and OR operations interact with each other. We have two of them:

$$x(y+z) = xy + xz$$
$$x + yz = (x+y)(x+z)$$

Remember the mathematician saying Boolean Algebra was a "distributive lattice"? This is what he means. Both operations distribute over each other. Our normal algebra only has one of these, because $3+4^*5 \neq (3+4)(3+5)$. In the structure called a lattice, as in studies of formal logic, we use the symbols $\vee$ and $\wedge$ to represent our operations. You'll see these sneak into engineering texts sometimes, but we're going to stick with + and $\cdot$ because we're more familiar with them and that familiarity will make our computations go more smoothly. It's easier to see the use of the distributive property in an expression of the form

$$\overline{\bar{x}y + \bar{x}\,\bar{y}} + xz = \overline{\bar{x}(y+\bar{y})} + xz$$

than in its mathematical logic equivalent

$$\neg((\neg x \wedge y) \vee (\neg x \wedge \neg y)) \vee (x \wedge z) = \neg(\neg x \wedge (y \vee \neg y)) \vee (x \wedge z)$$

if you are unaccustomed to these symbols.

We have another operation, NOT. To see how it interacts with AND and OR we turn to **De Morgan's Laws** (Fig. 2.4).

You can learn these mechanically by seeing that when you NOT an AND or OR you NOT each variable and then swap the operation (AND becomes OR and OR becomes AND.) Logically, you can reason it out by thinking that in order for x OR y to be 0 it must be the case that x = 0 AND y = 0 while in order for x AND y to be 0 we need only require one of x OR y to be 0. (Recall the expression $\bar{x}$ can be thought of as "x is 0".)

Remember in thinking through these to try to stick with 1's and 0's and stay far away from mixing in Trues and Falses and thinking about these as English statements: that way lies something called propositional calculus and it won't directly help us to design our digital systems and at times it may even lead us astray.

We can put all these rules together work through complex Boolean simplification problems. For example:

$$\overline{x+y} = \overline{x}\,\overline{y}$$
$$\overline{xy} = \overline{x} + \overline{y}$$

Fig. 2.4 De Morgan's Laws

$$\overline{\overline{x+y}} + \overline{(x\bar{y}+z)\overline{yz}}$$
$$x\bar{y} + (\bar{x}+y)\bar{z} + \bar{y} + \bar{z}$$
$$x\bar{y} + \bar{x}\bar{z} + y\bar{z} + \bar{y} + \bar{z}$$
$$\bar{y}(x+1) + \bar{z}(\bar{x}+y+1)$$
$$\bar{y} + \bar{z}$$

The second to last step occurs frequently enough that there is another set of laws, called **absorption properties**, to cover them:

$$x + xy = x$$
$$x + \bar{x}y = x + y$$

If you're doing this kind of work a lot it's useful to memorize these absorption rules. If not, they are easy to derive in the normal course of things: $x+xy = x(1+y) = x$ and $x + \bar{x}y = (x+\bar{x})(x+y) = 1(x+y) = x+y$. This is why the distributive properties and DeMorgan's Laws are fundamental while absorption rules and others like them are very much secondary and don't need to be able to be recited on demand when awoken at 3 AM.

Exercises

2.1 A security system has an active-low master switch (M) and active-high sensors (S1 and S2) as well as an active-high test button (T). The alarm (A) should sound (logic 1) when the master switch is active and either sensor is active or when the test button is pressed (even if the master switch is inactive.) Write a Boolean Algebra equation for A.

2.2 A control panel has two lights (L1 and L2) that turn on when logic-1 signals are present. L1 is on when an active high temperature sensor (T) is off or when either of a clock (C) or motion (M) signal are logic 1. L2 is on whenever the clock is logic-0 but the temperature sensor is on. Write Boolean Algebra equations for L1 and L2.

2.3 The controller for an automatic door at the grocery store will open the door (D is logic-1) when it detects activity on an active-low pressure sensor (P), when an override switch (R) is logic-1, or when the active-low safety stop signal is active (S). Write a Boolean Algebra equation for D.

2.4 A component in a computer's datapath needs to activate (logic-0) for MUL, ADD, MOV, and DIV instructions. The instruction set uses 4-bit opcodes (represented by $x_3x_2x_1x_0$). The opcode for MUL is 1011, ADD is 1100, MOV is 1010, and DIV is 1000. Write a Boolean Algebra equation for the control signal C which activates this component based on the opcode.

2.5 A system has active-high inputs X and Y and active-low inputs B and C. The output F should be a 1 when at least one of X and Y are active and no more than one of B and C are active.

Write a Boolean equation for F.

2.6 A home security system has an active-low master enable (E) switch as well as active-high light (L), video (V), and motion (M) sensors and an active-low call to the police (P). The police should be called whenever activity is detected on at least one sensor and the system is enabled. Write a Boolean equation for P.

2.7 A circuit takes a four-bit input $x_3x_2x_1x_0$ representing the months January (0001) through December (1100). The output T should be 1 when the input is a month consisting of 31 days and 0 otherwise. Write a Boolean equation for T.

2.8 A circuit takes a four-bit input $x_3x_2x_1x_0$ representing a binary number. The output P should be 1 if and only if the number is prime and the output D should be 1 if and only if the number is divisible by five. Write Boolean equations for each output. (See Chap. 4 on unsigned numbers for the basics on binary numbers if you haven't encountered them before.)

2.9 The control unit in a computer needs to activate an active-high signal s_1 when the three-bit opcode (given by $x_2x_1x_0$) is 100 and 110 and it needs to activate an active-low signal s_2 when the three-bit opcode is 011 or 101. Write Boolean equations for each signal.

2.10 A device inputs two three-bit binary numbers x and y ($x_2x_1x_0$ and $y_2y_1y_0$) and outputs a 1 when the number x is greater than the number y. Write a Boolean equation for the output F.

2.11 A device inputs a four-bit binary number $x_3x_2x_1x_0$ and generates three outputs: E is 1 when the input number is even, G is 1 when the input number is greater than 9, and L is 1 when the input number is less than 4. Write Boolean equations for the outputs E, L, and G.

2.12 A majority function will be 1 when an input contains more 1's than 0's in its representation.

Write the Boolean equation for the output F of a majority function which takes a 4-bit input $x_3x_2x_1x_0$.

2.13 Consider a controller for a lighting system. There is an active-high motion sensor (M) and an active-low sensor (S) that turns on when the level of light filtering into the space from outside is above a certain threshold. The system needs to turn on active-high internal lights (L) when the motion is detected and the outside light is low. Write a Boolean equation for the internal light signal.

2.14 A vehicle has active-high sensors to detect whether the indicator switch is in the right (R) or left (L) position and an active-low signal to detect whether the brake is pressed (B). There are three output signals for the rear light controls: left rear (LR), right rear (RR) and blinking (BL.) A logic-1 to LR or RR

indicates the light is on and a logic-1 to BL indicates the light is blinking if it's on (if BL = 1 and LR or RR are 0 then nothing happens.) The rear lights need to be on when the appropriate indicator switch is thrown (left or right) as well as when the brake is applied. The rear lights need to be held constant when the brake is applied and blink when the indicator switches are active but the brake is not applied. Write Boolean equations for the three outputs of this system.

2.15 Use the rules of Boolean Algebra to simplify the following expressions.

(a) $A(B + \bar{A}C)$
(b) $A\left(\overline{A+B}\right) + C + \left(\overline{A+C}\right)$
(c) $X + W\bar{X}\bar{Y}$
(d) $\overline{ABC} + \overline{\bar{B}\bar{C}}$
(e) $\bar{X}Y + \bar{Y} + W(Z + \bar{W}X)$
(f) $ab + a\bar{b} + \bar{a}\bar{b}$
(g) $\overline{\bar{a}\bar{b} + \bar{a}c + \bar{a}b + ac}$
(h) $\overline{xy(\bar{x} + \overline{y\bar{z}})}$
(i) $(a + \bar{a}\bar{b})\overline{(bc + a\bar{c})\bar{a}}$
(j) $\overline{(\bar{x}\bar{y} + z)} + z + xy + wz$
(k) $\bar{A}B + \bar{B} + C(D + A\bar{C})$
(l) $\bar{C} + A\left(\overline{B + \bar{C}}\right)$

2.16 Use factoring to simplify the following Boolean expressions given in canonical form.

(a) $\bar{x}yz + \bar{x}y\bar{z} + xyz$
(b) $\bar{A}\bar{B}\bar{C} + A\bar{B}C + ABC + AB\bar{C}$
(c) $\bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}C + AB\bar{C}$
(d) $\bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D} + AB\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}D + ABCD$

2.17 A cabin in the woods has active-high sensors for the door (D) and chimney (C) and an active- low sensor for the window (W). It also has an active-high master switch (M) and a light meter (L) which is logic-1 during the day and logic-0 at night. The active-high alarm A needs to sound when any of the three sensors are active, the master switch is active, and it is night. Write a Boolean Algebra equation for A.

2.18 A system's warning light (W) needs to light up (logic-0) when exactly one of two sensors A and B are active and a timer input (T) is active. A and B are active-high; T is active-low. Write a Boolean Algebra equation for W.

2.19 Let $F = \overline{\left(A\bar{B}D + C + \overline{CA}\right)(\alpha + 1)}$. Use DeMorgan's Law and Boolean Algebra identities to simplify this expression.

2.20 Use Boolean Algebra to simplify the logic functions

(a) $F = \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C}$
(b) $G = \overline{\bar{D}\bar{D}E}$
(c) $H = Y\left(W + X + \overline{\bar{Y} + \bar{Z}}\right)Z.$

2.21 Construct the operation XOR from the basic Boolean operations AND, OR, and NOT.

2.22 Using DeMorgan's Law, show that a NAND gate is a functionally complete set. That is, show algebraically how you can use a NAND gate as a universal gate to implement AND, OR, and NOT functions.

2.23 Use DeMorgan's Laws to give a simplified expression for the complement of the following logic functions:

(a) $F = \bar{A}BC + \bar{D}$
(b) $G = A(B + \bar{C})$

2.24 A three input AND-OR gate produces a TRUE output if both A and B are TRUE, or if C is TRUE. Complete the truth table for this gate.

2.25 A three-input OR-AND-INVERT gate produces a FALSE output if C is TRUE and A or B is TRUE. Otherwise it produces a TRUE output. Complete the truth table for this gate.

2.26 Consider a security system with three inputs: video, motion sensor, and master switch. The system needs to sound an alarm (that is, output a 1) when either the video or motion sensor inputs are 1 and the master switch is not thrown (its input would be 0 when not thrown.) Complete the truth table for this system.

2.27 Boolean Algebra has two distributive properties. Write the one that is unique to the algebra of logic and not found in the normal algebra of the real numbers.

2.28 Find the minimal sum of the following logic function: $f = (\bar{a} + ab)\overline{(b\bar{c} + a)}(\bar{a} + c)$

2.29 Use Boolean algebra to find the minimal sum of the function $F = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$.

2.30 A computer scientist will enjoy his walk across campus on sunny days on which he encounters no squirrels. He will also enjoy his walk any day he sees a cardinal, as well as on rainy days where there are squirrels and cows. Write a Boolean expression for his enjoyment E in terms of sun S, squirrels Q, cardinals C, and cows W.

2.31 A circuit has four inputs x_3, x_2, x_1, and x_0, representing the months January (0000) through December (1100). The output T should be TRUE when the input is a month consisting of 31 days. Write a Boolean equation for T.

2.32 A circuit has four inputs x_3, x_2, x_1, and x_0. These inputs represent a BCD number. Write an equation for the output V which will detect an invalid BCD input (see Chap. 9 for details on the BCD format if you have not encountered it before.)

2.33 Because it's the only location on campus where wine and other fine spirits are permitted, the Kirk Memorial needs to be equipped with the latest alarm system. This system is comprised of various switches and sensors located in the memorial and tied into the City of Kirksville Police Station. The main switch is located at the police station, and it is turned on at night after hours. When this switch is activated, anyone wanting to enter the memorial has to turn on a secret switch located near the front door before entering or the alarm will sound. Also, so the University president can visit the memorial after hours

when the police station switch is on, he can turn on the switch located in the president's office which de-activates the system and allows him to get in. Each memorial worker has access to a button that can be pressed if the precious stores are being robbed. This button will turn on the alarm unless the switch by the president's desk is turned on, which will deactivate the memorial workers' buttons. (This is in case the president wants to raid the wine himself.) Also, during the day when the police station switch is off, the spirits can be accessed without an alarm sounding if the switch by the president's desk is on.

Let the signals be defined as follows:

Police station switch—P (L)
President's desk switch—D (H)
Memorial door sensor—M (L)
Memorial front door sensor—F (L)
Secret switch at front door—S (H)
Memorial worker switch—W (L)

Write a Boolean equation for the Alarm (A). Note that the signals marked L are active-low, meaning they carry a logic value of 0 (or FALSE) when active. The signals marked H are active-high, meaning they carry a logic value of 1 (or TRUE) when active.

2.34 Write Boolean algebra equations for the functions F, G, and H based on the truth table given in Table 2.5.

2.35 Design a system controlled by three switches A, B, and C. The output is to be 0 when all three switches are unactivated (logic 0). Then, the output should change level any time a switch is activated. Write a Boolean Algebra equation for this system.

2.36 A three-input OR-AND-INVERT gate produces a FALSE output if C is TRUE and A or B is TRUE. Otherwise it produces a TRUE output. Write a Boolean Algebra equation for this device.

2.37 Use the three basic rules of Boolean algebra to simplify the following expressions.

Table 2.5 Truth table for exercise 2.34

A	B	C	F	G	H
0	0	0	1	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	0	0	1

(a) $F = A(B + \bar{A}C)$
(b) $F = A\left(\overline{A + B}\right) + C + \left(\overline{A + C}\right)$
(c) $F = X + W\bar{X}\bar{Y}$
(d) $F = \overline{ABC} + \overline{\bar{B}\bar{C}}$
(e) $F = \bar{X}Y + \bar{Y} + W(Z + \bar{W}X)$

2.38 A control panel has two active-low inputs A and B and two active-high inputs C and D. An active-high light (L) needs to be on when at least one of A and B are active and at most one of C and D are active. An active-low buzzer (Z) needs to sound when exactly one of the signals A, B, and C are active. Write Boolean algebra equations for the outputs L and Z.

2.39 Simplify the following two Boolean expressions using the rules of Boolean algebra. Show your work.

(a) $\boldsymbol{F = \left(\bar{A} + B\right)\left(\overline{\bar{A} + B\bar{C}}\right)}$
(b) $\boldsymbol{F = \overline{A\bar{C}} + \overline{(A + \bar{B})(\bar{D} + C)}}$

Chapter 3
Logic Function Synthesis

Now expert at the task of writing logic functions using the basic operators to describe digital systems, it's time to take advantage of a useful tool: the **truth table**.

As we recall, the truth table is not about truth, but at least it's a table, so we've got that going for us. The truth table is a very helpful way for us to specify functions that might be difficult to get at via the straightforward construction methods of the previous chapter.

For example, let's consider a "prime number detector" device that evaluates to 1 if the input signal is the encoding for a prime number and to a 0 otherwise. We will encode the input number using three variables together (to get eight possible encodings) and we write it $x_2x_1x_0$. We often use multiple signals in this way to represent numbers or other encodings, always taking care to number the bits starting with a 0 on the far right (the *least significant bit*) going up to the largest number on the far left (the *most significant bit*.) In this way, we combine three separate Boolean Algebra variables into one entity when taken together.

It's quite hard to just look at this problem and divine the correct combinations of input variables to organize around the ands, ors, and nots we have available to us as Boolean Algebra operators. In cases such as these, the truth table is of great help as we design the function.

As seem in Table 3.1, the left column represents the number while the input $x_2x_1x_0$ is shown to take on all eight possible 3-bit values, starting with 000 and ending with 111. We'll analyze this pattern and how it relates to number encodings in a later chapter. For now, we can rely on the information in the leftmost column. We see that the function F is logic-1 for all the primes: 2, 3, 5, and 7. We call the process of filling out the truth table for a logic function **specifying the function.**

The next question is "Great, we have the table. That's nice. But where is the equation? I must have the equation for F in terms of x2, x1, and x0 in order to design my system." This is an excellent question and the process of going from the truth table to the equation is called **synthesizing a logic function**.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_3

Table 3.1 Truth table of a generic logic function F

Number	x_2	x_1	x_0	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Normal Forms

Being mathematical objects, there are a lot of ways to represent logic functions. To begin, we're going to focus on two important forms of functions. These are called **normal forms** or **canonical forms.**

The first, and the most intuitive, is called the **Sum of Products (SOP)** form (mathematicians call it the Disjunctive Normal Form ... we'll stick with the engineers on this one.) We get this form by looking at the 1's in the truth table. We say to ourselves, "Self, my prime detector function F evaluates to a 1 when the input is 010 or 011 or 101 or 111." We can write the math as

$$\text{F} = \text{input is } 010 + \text{input is } 011 + \text{input is } 101 + \text{input is } 111$$

Now we just need to fill in the correct variables for the inputs. We see the input is 010 when x2 is 0 and x1 is 1 and x0 is 0. Remembering how to use and, or, and not from the previous section, we understand that the input is 010 when the expression $\overline{x_2}x_1\overline{x_0}$ is 1. Filling in all four terms in this manner gives us the Sum of Products canonical form of our function:

$$\text{F} = \overline{x_2}x_1\overline{x_0} + \overline{x_2}x_1x_0 + x_2\overline{x_1}x_0 + x_2x_1x_0$$

It is called the Sum of Products form because the terms are products connected by addition. The other canonical form is found by focusing on the 0's in the truth table and is the opposite of the SOP form. It's the **Product of Sums (POS)** form.

The logic for this form is a bit more trying than that for the SOP form. We have to think that "the function F is 1 when the input is not 000 and the input is not 001 and the input is not 100 and the input is not 110." We have to use the and connective here because otherwise we'd have a situation where we're claiming F can be 1 as long as the input is not, say, 000, which leaves open the possibility of input 001 evaluating to 1, which is not what we're after. So, for the SOP form we have to connect the terms with ands. Now, let's look a the terms themselves.

What does it mean to say "the input is not 110", say? Well, the input *is* 110 when x_2 is 1 and x_1 is 1 and x_0 is 0. So if any one of these inputs is different, the input fails to be 110. In words, this means that the input is *not* 110 when x_2 is 0 or when x_1 is

0 or when x_0 is 1. This is not the most straightforward logic ever (unless you have studied this before and already now the august Frenchman's law) so be sure to go through it and make sure you have it. This is an example of a paragraph in a technical book that may require 5–10 min of study in order to grok.

With that, we are now ready to display the Product of Sums canonical form. We look at the rows of 0's on the truth table, invert each variable, and OR them together. The POS form of the prime number detector function F is given by

$$\mathrm{F} = (x_2 + x_1 + x_0)(x_2 + x_1 + x_0)(\overline{x_2} + x_1 + x_0)(\overline{x_2} + \overline{x_1} + x_0)$$

Again, it's crucial you spend some time with this to be sure you have it. The SOP form's justification comes easily; the POS requires more effort.

These forms are important because we'll see later that the way we write a logic equation directly affects how much hardware is required to implement it physically and also influences how much power the device consumes. We can see that a function that has, say, only a few 1's and is mostly 0's would have a short SOP representation but require many terms in the POS, and one that has a lot of 0's but few 1's would be more easily written in the POS form. If we want to build efficient digital systems (and we do), we need to be aware of the multiple ways we can implement our finely crafted functions. These two canonical forms are the starting point for logic function synthesis.

Minterms and Maxterms

Since truth tables are large and unwieldly, we have shortcut ways to write functions specified in our tables. We refer to the rows of 1's as **minterms** and the rows of 0's as **maxterms.** Then we can express the SOP form as a **sum of minterms** and the POS form as a **product of maxterms.** We refer to the rows by the number they represent (given in the prime number detector table to the left) and would write our function F as follows:

$$\text{Sum of Minterms: } F = \sum (m_2, m_3, m_5, m_7) = \sum (2, 3, 5, 7)$$
$$\text{Product of Maxterms: } F = \prod (M_0, M_1, M_4, M_6) = \prod (0, 1, 4, 6)$$

This is a much more compact way to express a logic function and will be used often hereafter.

In case you were wondering (and you should be wondering) why the rows with 1's are called minterms and the rows with 0's maxterms when it seems it should be the other way around, the reason is that these words refer to the number of terms required to make the function evaluate to logic-1, not to the supposed largeness of 1 compared to 0 (remember, in Boolean Algebra we don't have a comparison operation and saying $1 > 0$ doesn't actually make any sense, so there is no real reason to consider logic-1 to be "larger" than logic-0 at all.) The reason is that in the

Sum of Minterms form we require a *minimum* number of the terms (that is, we require just one term) to be logic-1 in order for the entire function to evaluate to logic-1. In contrast, in the Product of Maxterms form we require a *maximum* number of the terms (that is, we require all of them) to be logic-1 in order for the entire function to evaluate to logic-1. So, there you go. Minterms and maxterms.

We can do some cute logic tricks with minterm and maxterms. Review the following statement and make sure you understand (assuming F is a function of three variables):

$$\text{If } F = \sum(1,3,4) \text{ then } F = \prod(0,2,5,6,7) \text{ and } \bar{F} = \sum(0,2,5,6,7) = \prod(1,3,4).$$

Don't Cares

There is one more wrinkle that occurs when synthesizing logic functions. Right now our prime number detector only accepts inputs in the range 0 through 7. Let's expand this input range to allow input of any digit 0 through 9. To accommodate this, we are going to need to add another variable to our input so F is now a function of four variables (when encoding numbers using variables that can take on only two values, we can only work with up to 2^n different numbers when we have n variables. Since $2^3 = 8$ and we want ten inputs, we need to use four variables instead of three.)

So, our truth table now requires 16 rows!

Let's examine the 16-row truth table seen in Table 3.2. We don't actually get more 1's in the truth table because expanding input numbers to 9 doesn't introduce

Table 3.2 A 16-row truth table

Number	x_3	x_2	x_1	x_0	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	d
11	1	0	1	1	d
12	1	1	0	0	d
13	1	1	0	1	d
14	1	1	1	0	d
15	1	1	1	1	d

any more prime numbers. But now we are faced with an interesting dilemma: what to do about the last six input combinations 1010 through 1111. These correspond to decimal numbers 10 through 15, but in this application we don't care about these numbers. We only care about the digits 0–9. How do we represent the function in the truth table form when we have inputs we don't care about? The answer is simple and awesome: we introduce a new symbol d to represent that we don't care. We call these ***don't cares*** (yes, that's the honest-to-goodness technical term for them) and functions with don't cares are said to be **incompletely specified functions**.

You can see the six don't cares in the truth table. When we are specifying the function, we really don't care whether these entries are 1 or 0. When we physically implement the function we have to assign one or the other value to these entries, but at the specification stage we don't care.

The existence of don't cares doesn't much affect the SOP and POS canonical forms, but we do take care to note them when writing the minterm or maxterm forms. The expanded four-variable prime number detector complete with don't cares can be written as follows:

$$F = \sum (2,3,5,7) + d(10,11,12,13,14,15) \text{ or}$$
$$F = \prod (0,1,4,6) + d(10,11,12,13,14,15)$$

Don't get too caught up in the + in these forms as it's basically an abuse of notation for convenience sake. These are just nice ways to represent a full truth table for an incompletely specified function.

Day of the Week Detector

The previous example is not an outlier: we'll quite often, if not **usually**, have *multi-bit* inputs or outputs where we need to use multiple Boolean Algebra variables to represent a single quantity. In this example we want to develop a device that will take as input a day of the week and assert logic-1 whenever the input is Wednesday, Saturday, or Sunday. Since there are more than 2 days of the week, we have to use multiple input variables to represent the single input. In this way we have multiple variables combined into a cohesive whole. Sometimes this gets confusing because while we think about them as independent entities as a designer in reality from the point of view of our digital system we need to encode them all as individual variables.

To see this, we first need to consider that we'll need three bits to represent all 7 days of the week. With one variable we can represent two things, with two variables four things, and it takes three variables to encode up to eight things. Since seven is greater than four, we have to use three bits even though we'll have one encoding left over.

Let's say we choose to associate our inputs with days of the week in Table 3.3.

Table 3.3 Days of the week assigned to consecutive binary values

X_2	X_1	X_0	Day
0	0	0	Sunday
0	0	1	Monday
0	1	0	Tuesday
0	1	1	Wednesday
1	0	0	Thursday
1	0	1	Friday
1	1	0	Saturday
1	1	1	

We see that we have one bit pattern, 111, which is unused. We can fill it in with don't cares.

In order to design the device, we need to write Boolean Algebra equations for the output in terms of each of the three inputs. It's awkward because we're thinking conceptually of the input as a whole while we have to design the device with the view of the individual variables x_2, x_1, and x_0 that together constitute the input. But, this is where the truth table helps guide our thinking. Once we've built the table all we have to do is read off the SOP or POS form to get the correct combinations of ANDs, ORs, NOTs, and $x_i's$ necessary to implement our device.

So, for this example, we want to pick off the rows of the table corresponding to Wednesday, Saturday, and Sunday. So the canonical SOP form would be

$$F = \overline{x_2}x_1x_0 + x_2x_1\overline{x_0} + \overline{x_2}\ \overline{x_1}\ \overline{x_0}$$

and the canonical POS form would be

$$F = (x_2 + x_1 + \overline{x_0})(x_2 + \overline{x_1} + x_0)(\overline{x_2} + x_1 + x_0)(\overline{x_2} + x_1 + \overline{x_0})$$

At first glance, it appears the SOP form has fewer terms and therefore would be easier to implement physically. However, to be sure we'd have to perform further analysis to **minimize** the logic function to see which form, sum or product, is ultimately the best to use. The presence of the don't cares complicates this process and makes it even harder to see from the truth table or the canonical form (because these forms don't even show us the don't cares we can't really work with them yet.) The next chapter shows how basic logic function minimization works and gives procedures to decide among competing implementation formulas.

Building a Normal Form

One helpful tool in the Boolean Algebra toolbox is the ability to construct a normal form from simpler expressions. To do this, we either multiply a product term by 1 (useful for building the SOP form out of another sum) or add 0 to a sum term (useful for building the POS form out of another product):

Multiply product by 1 **(SOP)** $xy = xy1 = xy(z+\bar{z}) = xyz + xy\bar{z}$
Add 0 **to sum (POS)** $x+y = x+y+0 = (x+y)+z\bar{z} = (x+y+z)(x+y+\bar{z})$

Of these, the first is the easiest for us to see because we're so used to the fact that multiplication distributes over addition. The second relies upon the other, Boolean Algebra specific, property that says addition distributes over multiplication (which doesn't hold for real numbers.)

Let's do two examples. First, assume we have $F = \overline{x_2}x_1 + x_1x_0 + x_2\overline{x_1\ x_0}$ and we want the canonical SOP form of the function. The last term already contains all three variables so we just need to introduce variables to the first two terms. Following our "multiply product by 1" rule, we get

$$\begin{aligned} F &= \overline{x_2}x_1 + x_1x_0 + x_2\overline{x_1\ x_0} \\ &= \overline{x_2}x_1(x_0 + \overline{x_0}) + x_1x_0(x_2 + \overline{x_2}) + x_2\overline{x_1x_0} \\ &= \overline{x_2}x_1x_0 + \overline{x_2}x_1\overline{x_0} + x_2x_1x_0 + \overline{x_2}x_1x_0 + x_2\overline{x_1x_0} \\ &= \overline{x_2}x_1x_0 + \overline{x_2}x_1\overline{x_0} + x_2x_1x_0 + x_2\overline{x_1x_0} \end{aligned}$$

In the final step we are eliminating a duplicate term. This can easily happen during this process and you should always check for that once all terms have been expanded.

Now, for the POS example, let $F = (x_1 + \overline{x_0})(\overline{x_2} + x_1)$. In this case we follow our "add 0 to sum" rule to get

$$\begin{aligned} F &= (x_1 + \overline{x_0})(\overline{x_2} + x_1) \\ &= ((x_1 + \overline{x_0}) + \overline{x_2}x_2)((\overline{x_2} + x_1) + \overline{x_0}x_0) \\ &= (\overline{x_2} + x_1 + \overline{x_0})(x_2 + x_1 + \overline{x_0})(\overline{x_2} + x_1 + \overline{x_0})(\overline{x_2} + x_1 + x_0) \\ &= (\overline{x_2} + x_1 + \overline{x_0})(x_2 + x_1 + \overline{x_0})(\overline{x_2} + x_1 + x_0) \end{aligned}$$

Again, as in the previous example, we find ourselves with a duplicate term in our canonical form and we eliminate it. This derivation is harder to see right off the bat than the SOP form because it uses the unintuitive distributive property $x + yx = (x+y)(x+z)$ that we need to get used to if we are to work with digital systems. One reason to work a lot of the symbolic mathematics problems is to help us intellectually grasp the underlying ways in which our digital components interact as described by the Boolean Algebra. It's not that we need to do all these calculations by hand in our day-to-day job as digital designer; rather, it's that by doing this work we develop our minds to the point where we can see farther and design better.

Because it's sometimes useful to have the normal form instead of the simplified form, and because it's usually always better to have the normal form instead of some intermediate form that is unrecognizable as something we tend to want to work with, this technique is one you should be very familiar with and ready to employ when the situation demands.

Exercises

3.1 For each of the following logic functions, write the truth table and the SOP and POS canonical forms.

(a) $F(a,b,c) = ab + \bar{a}c(b + \bar{c})$
(b) $F(A,B,C,D) = \prod M(0,1,4,5,7,8,9,15)$
(c) $F(x,y,z) = \sum m(0,2,4,5,6)$

3.2 For this problem, consider the following functions:

$$F = \sum m(1,2,3,7)$$
$$G = \prod M(0,2,4,5)$$

(a) Write F as a product of maxterms.
(b) Write G as a sum of minterms.
(c) Write $\bar{F}$ as a product of maxterms.
(d) Write $\bar{G}$ as a sum of minterms
(e) Write $\bar{F}$ as a sum of minterms.

3.3 Give the canonical SOP form of the logic function $F(A,B,C) = \prod M(1,2,5,6,7)$

3.4 Write the SOP and POS forms of the logic function $F(X,Y,Z) = \sum m(0,1,5,6)$

3.5 Give the SOP form of the function $F(A,B,C) = \prod M(0,1,5,6,7)$.

3.6 Give the complement of the logic function $F(A,B,C,D) = \sum m(0,4,5,6,9,11,14,15)$ in both minterm and maxterm forms.

3.7 Find the complement of the logic function $F(X,Y,Z) = \sum m(1,3,4)$

3.8 Find the complement of the logic function $F(A,B,C,D) = \prod M(7,9,10,11,15)$

3.9 Write the canonical POS form for the following 3-variable logic function: $F = \sum(1,2,4,7)$.

3.10 Write the canonical SOP form for the following 3-variable logic function: $F = \prod(0,3,5,6)$.

3.11 Write the POS form of the logic function F if $\bar{F} = \prod(1,3,6)$.

3.12 Design an Interrupt priority encoder. It should be able to handle inputs from 4 different devices, numbered 1 through 4, and output the binary representation of the decimal number of the device. Priority should be assigned such that the device with the lowest number has priority. If no input line is asserted, then the output should be 0.

3.13 Fill in the truth table for the following three-variable logic function: $F = A\bar{B} + \overline{B \bigoplus C}$.

3.14 Fill in the truth table for a logic function which is 1 if the input corresponds to a digit in the set {0, 1, 2, 6, 9} and 0 otherwise. Assume input in XS3 format (See Chapter 9 for the details of XS3 format if you are unfamiliar with it.) Find the canonical SOP and POS forms of the function.

Table 3.4 Truth table for exercise 3.16

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

3.15 Suppose a function F is equal to 1 if the 3-bit unsigned binary input is equal to 2, 3, 4, or 7. Write the logic table for this function and calculate the Sum of Products (SOP) canonical form.

3.16 Consider the logic function defined by the truth table given in Table 3.4. Find the SOP and POS forms of F. Also write the minterm and maxterm decimal representations of F.

3.17 If $F = \sum m(0, 2, 3, 6, 7)$, find the algebraic SOP form of $\bar{F}$.

3.18 For each of the following logic functions, write the truth table and write the SOP and POS canonical forms

(a) $F(a, b, c) = ab + \bar{a}c(b + \bar{c})$
(b) $F(a, b, c, d) = \sum m(0, 1, 4, 5, 7, 8, 9, 15)$
(c) $F(x, y, z) = \sum m(0, 2, 4, 5, 6)$
(d) $F(w, x, y, z) = \sum m(1, 2, 3, 4, 5, 7, 12, 13, 15)$
(f) $F(a, b, c, d) = \prod M(4, 6, 7, 11, 12, 14, 15)$

3.19 Consider a robot with input sensors for obstacle left (L), obstacle right (R), proximity to source (P) and low battery (B). It has outputs for move right (MR), move left (ML), and return to source (S). The robot should move around a given area while avoiding obstacles and staying close to the source. When it indicates an obstacle to the right or left (R or L are logic 1) then the robot should move in the opposite direction to avoid (set MR or ML to logic 1 accordingly.) When it is no longer in proximity to the source (P = 0) or when the battery is low (B = 0) then it should return straight to the source (set S = 1) while trying to avoid obstacles on the way.

3.20 Draw the truth table for the logic controller for this robot and write the SOP and POS canonical forms for all three output variables.

Xerxes, Yezabel, and Zebulon are students whose class schedules are as follows:

Xerxes: 9:30—10:20, 2:00—2:50 MWF, 2:00—3:15 TR
Yezabel: 8:30—9:20, 2:00—2:50 MWF, 12:00—1:15 TR
Zebulon: 9:30—10:20 MWF, 12:00—1:15, 3:30—4:45 TR

Define the following variables:

X = 1 if Xerxes goes to all his classes today and X = 0 if Xerxes skips all his classes today
Y = 1 if Yezabel goes to all her classes today and Y = 0 if Yezabel skips all her classes today
Z = 1 if Zebulon goes to all his classes today and Z = 0 if Zebulon skips all his classes today
D = 1 if today is MWF and D = 0 if today is TR

(a) Define a function $F(X, Y, Z, D)$ which is 1 if at least two of the students are in class at the same time today and 0 otherwise.
(b) Fill in the truth table for F, write the SOP and POS canonical forms for F, and use Boolean algebra to simplify the SOP form.

3.21 There are four adjacent parking spots in a particular parking area. There is a sensor mounted on each spot whose output is equal to 0 when a car is occupying the spot and equal to 1 otherwise (Note: we call this "active low" logic because logic-0 is asserted when something happens). Design a decoding system which will generate a 0 output if and only if there are two or more adjacent vacant spots available. Draw the truth table for this problem. Include a diagram of the parking spots labeled with the variables you use in your truth table. Find the canonical SOP form for your logic function.

3.22 Many computer instruction sets have an instruction such as INC A which increments (adds 1 to) the value in register A. Design a device which will perform an increment operation on a 3-bit signed input (See Chap. 8 for details on two's complement representation if you haven't encountered it before, or just ask your instructor if you can use unsigned inputs instead.). Fill in the truth table and find the canonical SOP representation for the outputs. You should have three inputs and three outputs (one for each bit of the sum.)

3.23 For the Boolean function described by the truth table given in Table 3.5, give the SOP, POS, minterm, and maxterm forms.

Table 3.5 Truth table for exercise 3.23

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

3.24 Design a device which will read in an instruction and output the control signals necessary to execute the instruction. Assume a very simple processor with six possible instructions: ADD, MUL, DIV, LD, ST, MOV.

The instructions are each given a 3-bit binary code (called the "opcode") as follows: ADD = 010, MUL = 110, DIV = 001, LD = 111, ST = 101, MOV = 011.

The three possible control signals are s_2, s_1, and s_0. We assume active high logic, so when a signal is active it is equal to 1 and when it is inactive it is equal to 0. Each instruction requires a certain combination of control signals to be activated: ADD activates s_2 and s_0, MUL activates s_0, DIV activates s_1 and s_0, LD activates s_2 and s_1, ST activates s_2, and MOV activates s_1.

Draw the truth table for this device. Your input is the 3-bit opcode and your output functions are the three control signals. Output all 0's for the ununsed opcodes.

3.25 A home security system has a master switch that is used to enable an alarm, lights, video cameras, and a call to local police in the event one or more of six sensors detects an intrusion. In addition there are separate switches to enable and disable the alarm, lights, and the call to police. The inputs, outputs, and operation of the logic required to control this system are as follows:

Inputs

S = signals from sensors (0 = intrusion detected; 1 = no intrusion detected)
M = master switch (0 = entire security system enabled; 1 = security system disabled)
A = alarm switch (0 = alarm disabled; 1 = alarm enabled)
L = light switch (0 = lights disabled; 1 = lights enabled)
P = police switch (0 = police call disabled; 1 = police call enabled)

Outputs

A = alarm (0 = alarm on; 1 = alarm off)
L = lights (0 = lights on; 1 = lights off)
V = video cameras (0 = cameras off; 1 = cameras on)
C = call to police (0 = call off; 1 = call on)

Operation

If any of the sensors detect intrusion and the security system is enabled, then outputs activate based on the settings of the remaining switches (A, L, and P.) Otherwise, all outputs are disabled.

3.26 Give the complement of the logic function $f(w,x,y,z) = \sum m(0,4,5,6,9,11,14,15)$ in both minterm and maxterm forms:

3.27 Write the truth table for each of the following functions:

(a) $F(a,b,c) = ab + \bar{a}c(b + \bar{c})$
(b) $F = A + \overline{BC}$
(c) $F = XY + (\bar{X} + W\bar{Y})$
(d) $F = B + A\bar{B}C + \overline{AC}$

3.28 Use a truth table to design a system controlled by three active-high switches A, B, and C. The output is to be 0 when all three switches are unactivated. Then, the output should change level any time a switch is activated. Fill in the truth table and write the output equation for this system.

3.29 Design a device which inputs a 4-bit binary number $x_3x_2x_1x_0$ and outputs a three-bit number $z_2z_1z_0$ equal to half the input number, rounding down.

3.30 A three-input OR-AND-INVERT gate produces a 0 output if C is 1 and A or B is 1. Otherwise it produces a 1 output. Complete the truth table for this gate.

3.31 A three input AND-OR gate produces a 1 output if both A and B are 1, or if C is 1. Complete the truth table for this gate.

Chapter 4
Basic Logic Function Minimization

Now that we have the SOP and POS canonical forms of our logic functions available to us, it's worth asking ourselves if we can investigate other, simpler, forms. Remember that these equations are essentially the blueprints for the physical device that's going to be implementing the functions and the more complicated the equations the less efficient the final product is going to be. Therefore, it's worth considering trying to find the "best" form of the function. We call this the **minimal form** and the process of finding this form is **logic function minimization**.

This section details some basic methods that every digital designer should be able to say they've worked through a few times. The next section contains more advanced methods that we'll call upon at certain points throughout the text but that can be safely skipped for the most part if you're ready to get moving to the next topic.

Factoring the Sum of Products

It's important to note at the outset that in professional digital design work the minimization is done almost entirely by software. The digital designers **specify the functions** as discussed in the previous section and then let the computers do the minimization. Don't think that Intel or NVIDIA have armies of interns sitting around in dark offices calculating minimal forms using the methods of this section. The value in spending time minimizing logic functions by hand is pretty much entirely that you, as a computing professional, will gain a more solid understanding of how Boolean Algebra and our logic operations work and that will help you see some interactions more clearly as a designer (such as when you can quickly reduce certain physical circuit configurations to others.) It's basically eating your unsavory but nutritious vegetables.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_4

Let's recall the SOP form of our prime number detector from the previous section:

$$F = \overline{x_2}x_1\overline{x_0} + \overline{x_2}x_1x_0 + x_2\overline{x_1}x_0 + x_2x_1x_0$$

Now, how can we reduce the number of terms in this function? Well, we turn to our beloved old friend from our high school or college algebra days: factoring! It turns out that the *distributive property* (e.g., the fact that $x(y+z)=xy+xz$) is stil valid in Boolean Algebra! This is the rule that lets us factor. So we can, say, look at the first two terms in our SOP form and notice that we can factor out the $\overline{x_2}x_1$ terms and get

$$F = \overline{x_2}x_1\left(\overline{x_0} + x_0\right) + x_2\overline{x_1}x_0 + x_2x_1x_0$$

How does this help? Well, since we have a two-valued logic system we know that either $\overline{x_0}$ or x_0 must be 1 and therefore the sum $\overline{x_0} + x_0$=1. This is a key fact. Make sure you spend some time with this line until you get it. It's going to come up over and over again.

Our equation is now $F = \overline{x_2}x_1 + x_2\overline{x_1}x_0 + x_2x_1x_0$, which is reduced from the canonical form. We're making progress! We see that the last two terms can also be grouped together for factoring:

$$F = \overline{x_2}x_1 + x_2x_0(\overline{x_1} + x_1) = \overline{x_2}x_1 + x_2x_0 1 = \overline{x_2}x_1 + x_2x_0.$$

We now have a simpler equation and there is no more factoring available to us, so we declare this is a **minimal sum form** of the function. (Yes, it's *a* minimal sum form, not *the* minimal sum form because, math being math, there can be *multiple* equivalent minimal sum forms of some functions. Just what you wanted to hear, I know.)

Let's review what we just did: we grouped terms for factoring so we had a bunch of variables and their complements so we could use the fact that $x + \bar{x} = 1$ to reduce the terms. That's it—the trick is to look for terms to group that differ in only one variable. That way we can eliminate that variable from the two terms and reduce the equation. It doesn't help to factor if we can't use the $x + \bar{x} = 1$ rule to reduce, so we are always looking for what we call *one-bit differences*; that is, looking for terms that differ in exactly one variable.

Another example:

$$F = \sum(1,3,4,7) = \overline{x_2}\,\overline{x_1}x_0 + \overline{x_2}x_1x_0 + x_2\overline{x_1}\,\overline{x_0} + x_2x_1x_0$$

What terms can we group together? Where are the *one-bit differences*? We see the first two terms, minterms 1 and 3, can be grouped and the x_1 term factored out and eliminated. But we also notice that the second and last terms, minterms 3 and 7, can likewise be grouped and the x_2 term factored out and eliminated. So which do we choose? Do we group minterm 3 with minterm 1or with minterm 7? Stop reading and think about it for a second, because the answer is awesome.

Remember that this is no longer algebra on the real numbers. We are no longer in a world where exotic constructs such as "2" exist. This means that $x + x = 2x$ is an equation that has no meaning to us. What, then, does $x + x$ equal? Well, it must equal x, because if x is 0 we get $0 + 0 = 0 =$ x and if x is 1 we get $1 + 1 = 1 =$ x. So, in Boolean Algebra $x + x = x$. Now, how does this fact help us in our dilemma? How can this help us decide how to group poor minterm 3, torn between 1 on one side and 7 on the other? It helps because we can simply double up minterm 3 and let it group with *both* minterms 1 and 7!

Using our new fact, we can write F as

$$F = \overline{x_2}\ \overline{x_1} x_0 + \overline{x_2} x_1 x_0 + \overline{x_2} x_1 x_0 + x_2 \overline{x_1}\ \overline{x_0} + x_2 x_1 x_0.$$

We have doubled up on minterm 3 and it's now clear to us that we can group both minterms 1 and 3 as well as minterms 3 and 7 to eliminate both x_1 and x_2 from the terms.

We now have that $F = \overline{x_2} x_0 + x_1 x_0 + x_2 \overline{x_1}\ \overline{x_0}$. (Work through the steps and make sure you follow this—it's not trivial.) Minterm 4 won't group with any of the remaining terms so there is no more factoring to be done and we've arrived at our minimal sum.

To review: to find the minimal sum from the SOP form we group terms with one-bit differences, factor, and eliminate variables. We stop when no one-bit differences remain.

This works pretty well for three-variable functions, but gets tedious really fast as we move up to four and five variable functions. Consider the function $F = \sum(2, 3, 5, 7, 8, 10, 12, 14, 15)$ and consider how much scanning and factoring and staring at bars and subscripts is required to find its minimal form.

There must be a better way.

A Visual Aid to Factoring

To find this better way, let's focus in on the core repetitive process, what computer scientists might call the *basic operation* of this minimization algorithm (yes, we've described an algorithm here.) The main thing we do is look for one-bit differences. Therefore, to expedite the process, let's make that easier.

We note that the one-bit differences we are looking for in the algebra are the same one-bit differences we see in the truth table itself. Minterm 1 is input 001 and minterm 3 is input 011. We can just compare the 1's and 0's of the inputs directly, skipping the algebra completely, and see that we can group minterms 1 and 3 because of the one-bit difference in the middle bit. That middle bit is the x_1 term so we know it drops out when we factor these minterms and we must be left with the remaining two, with x_2 being 0 and x_0 being 1. Thus, the resulting term is $\overline{x_2} x_0$.

Table 4.1 Truth table for $F = \sum(1, 3, 4, 7)$

Number	x2	x1	x0	F
0	0	0	0	**0**
1	0	0	1	**1**
3	0	1	1	**1**
2	0	1	0	**0**
4	1	0	0	**1**
5	1	0	1	**0**
7	1	1	1	**1**
6	1	1	0	**0**

Fig. 4.1 Reorganized truth table for $F = \sum(1, 3, 4, 7.)$

		X_1X_0			
		00	01	11	10
X_2	0	0 (0)	1 (1)	1 (3)	0 (2)
	1	1 (4)	0 (5)	1 (7)	0 (6)

What if we restructured the truth table to help us spot these groups more easily? What if we ordered the truth table in a way that will result in the one-bit differences appear *next to each other* visually on the table? After all, we don't have to follow the 000, 001, 010, 011, ... organization. Why not choose a different order? An order that emphasizes one-bit differences?

It turns out this sort of ordering is very important in computing in general, so it's worth considering this point in some detail. Table 4.1 shows the resulting table for our function $F = \sum(1, 3, 4, 7)$.

We've mixed up the order of the minterms, but the information Table 4.1 conveys is exactly the same as the table in our normal order (or in any order, really.) But what we now see is that the minterms 1 and 3 are next to each other on the table and easy to see visually that they group!

But wait, what about minterms 3 and 7? They are still not next to each other? I guess we could put 7 after 3, but then we'd still have gaps elsewhere. Despite the promise this method shows, no matter how cleverly we arrange the minterms in this table, there will *always be one-bit differences that are not next to each other*.

What do we do? Do we cry or give up? No, we do neither. We watch the *Matrix* movies—all three of them, even the bad ones—and let our imaginations expand. Why limit ourselves to a one-dimensional table, when we can have a *map*! Consider the arrangement of the truth table displayed in Fig. 4.1.

This version of the truth table has all the same information as our other version: we still have eight labeled cells with function values in them. The difference is that the cells are arranged in two rows and you can tell what input combination corresponds to which cell by its coordinate. In the top row all four cells have $x_2 = 0$ while in the bottom row each cell has $x_2 = 1$. In the third column both cells correspond to inputs where x_1 and x_0 are both 1. The number in the lower right corner tells us which minterm the cell represents. Go through this map, compare it to the table, and make sure you see how the coordinates work.

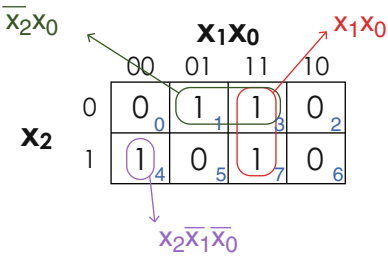


Fig. 4.2 Groupings for $F = \sum(1, 3, 4, 7)$

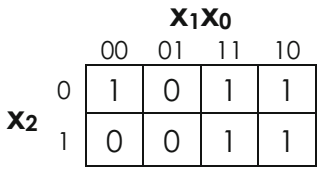


Fig. 4.3 K-map for $F = \sum(0, 2, 3, 6, 7)$

The thing that makes this arrangement of function values useful is that now we are ensured that every one-bit difference is visually apparent. All the one-bit differences are in adjacent cells and this makes the groupings obvious. See how we finally have minterms 3 and 7 as well as minterms 3 and 1 next to each other?

To use this map we group each minterm in as large a grouping as possible. Since a grouping represents factoring two terms in the SOP form, we must only use groups of 2 or 4 or 8 or 1 (powers of 2.) We cannot group 3 or 5 or 7 cells together.

Figure 4.2 shows what the map looks like after our grouping is completed.

One grouping represents factoring minterms 1 and 3 to arrive at the reduced term $\overline{x_2}x_0$. You can tell that is the result on this map because the x_1 term changes from 0 to 1 within the grouping, thereby indicating that both terms $\overline{x_1}$ and x_1 are present and ready to be eliminated using our $x + \bar{x} = 1$ rule.

The same process applies to the next grouping, yielding x_1x_0 because we can see that the variable x_2 is 0 in the top cell and 1 in the bottom cell, therefore changing within the grouping and dropping out of the equation.

The final term is a singleton grouping because minterm 4 does not share any one-bit differences with the other terms of the function.

So, that's it! We can use this clever map to visually identify the groupings necessary to reduce a function from SOP form into a minimal form while bypassing any of the algebra. This structure is called a **Karnaugh Map,** or commonly **K-map** for short, after its inventor, Maurice Map.

For another example, using logic function $F = \sum(0, 2, 3, 6, 7)$, see Fig. 4.3

We can see directly from the K-map that we have a grouping of four among minterms 2, 3, 6, and 7. What we would find if we worked through the algebra is that after grouping 2 and 3 together and factoring, and grouping 6 and 7 together and factoring, the resulting reduced terms would still have a one-bit difference and we could factor again! For every power-of-2 cells in a grouping we reduce one

variable from the group. Since 4 is 2^2 we reduce two variables and are left just with x_1 remaining (both x_2 and x_0 carry different values in different cells within the grouping, so they are both eliminated.)

It looks like the minterm 0 may be without another term to group, but we know that the minterm 4 has a one-bit difference with 0. Therefore, we need to view this map not as a two-dimensional thing on the paper but as wrapping around where the cells in the first column can group with the cells in the fourth column. Whoa, I know. So, we can group minterms 0 and 4, again recognizing visually that they can be factored. The need for algebra is replaced by the playing of a strategy game of grouping 1's. The final map after groupings is shown in Fig. 4.4.

You can even use K-maps to quickly see multiple equivalent minimal forms for functions that have more than one. Consider the function $F = \sum(0, 1, 5, 6, 7)$ and its map seen in Fig. 4.5.

We can see minterm 0 must be grouped with minterm 1 and minterm 7 must be grouped with minterm 6. However, minterm 5 can be grouped with *either* minterm 2 or 7. Each choice gives a different minimal form but the two possible forms are equivalent. Be sure not to include both groupings 2–5 and 7–5, however. While the resulting equation will be equal mathematically to the function, it will have an extra term and not be a minimal form. A common error when working with K-maps is actually including *too many* terms. If you are going to spend lots of time with these, that is a skill to develop.

What about incompletely specified functions? The ones with don't cares? Let's check out a map for one of these. Consider $F = \sum(0, 3, 5) + d(1, 6, 7)$. Its K-map can be seen in Fig. 4.6.

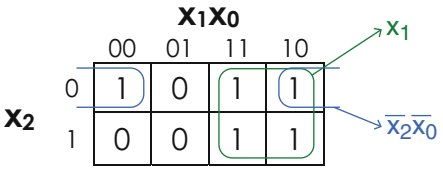


Fig. 4.4 Groupings for $F = \sum(0, 2, 3, 6, 7)$

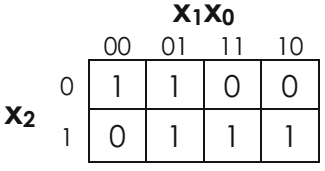


Fig. 4.5 K-map for $F = \sum(0, 1, 5, 6, 7)$

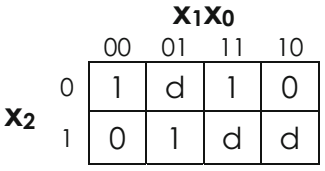


Fig. 4.6 Incompletely specified K-map for $F = \sum(0, 3, 5) + d(1, 6, 7)$

The rule here is that since the specification doesn't care about the value in the *d* cells, when it comes time to minimize the function we can choose to let the don't cares be either 1 or 0 depending on which helps us minimize the function the best. Here we see the misnomer that is the term *don't care*: while we truly don't care in the *specification* of the function, we truly *do very much care* at the implementation/minimization stage of building the device. So maybe they could be called *don't care insofar as the specification goes* or *don't cares until we're caging lightning to run the thing*, but the technical term *don't care* is what we've got. We'll just have to roll with it.

Back to the map, we see that if we let the don't cares at minterms 1 and 7 be 1's we get a nice 4-grouping as shown in Fig. 4.7.

We might be tempted to try to group the *three* minterms 0, 1, and 3 together but as we keep in mind that the groupings represent factoring algebraic terms by two's we come to our senses and shy away from trying to build a non-power-of-2 group.

The don't care at minterm 6 is chosen to be a 0 because it doesn't help us with any other grouping. If we make it a 1 we have to circle it as a singleton and that doesn't give us the minimal form.

So, the rule of don't cares is to use them when they help and ignore them when they don't help.

K-maps scale up as you add variables to your functions. Figure 4.8 shows a four-variable example: $F = \sum(1, 3, 5, 7, 11,12,13,15)$.

We see six 1's right in the middle and get excited and want to group them all, but we must remember to only group in powers of two. The trick is to get each grouping to be as large as possible while also having as few groupings as we can. It's OK to use a given cell multiple times, but each grouping needs to have at least one unique cell. If it's entirely subsumed by one or more other groupings then it is not relevant

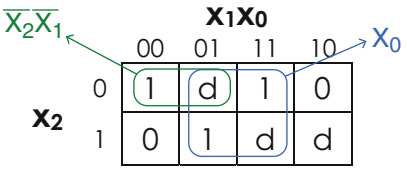


Fig. 4.7 Groupings for $F = \sum(0, 3, 5) + d(1, 6, 7)$

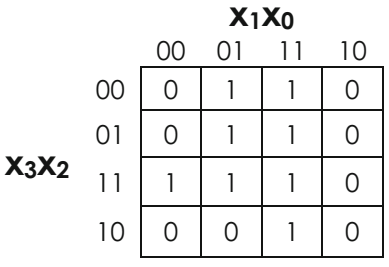
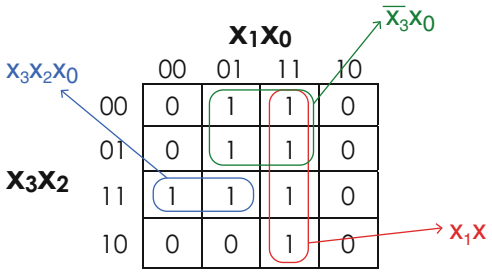


Fig. 4.8 K-map for $F = \sum(1, 3, 5, 7, 11,12,13,15)$



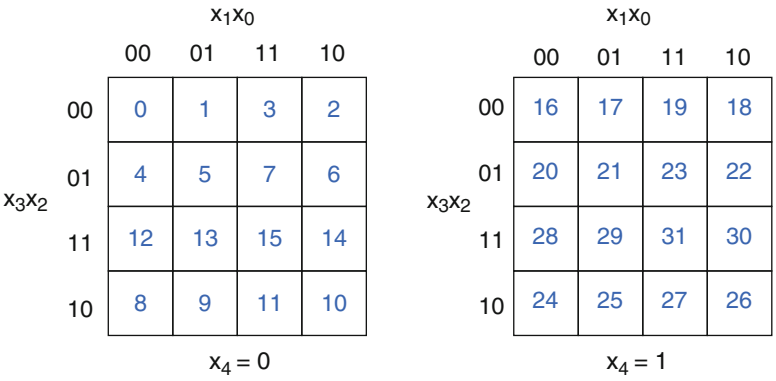


Fig. 4.9 Groupings for $F = \sum(1, 3, 5, 7, 11, 12, 13, 15)$

Fig. 4.10 A five-variable K-map

to the minimal form. The finalized K-map for the above four-variable function is given in Fig. 4.9.

But why stop there? Why only four variables? Let's see!

5- and 6-variable K-maps

If the exposition thus far has excited you, then this section is for you. In the previous section we worked with 3 and 4 variable maps. We saw that the size of the map doubles each time you add a variable to the mix. What happens if you have five variables? Well, we end up with a map twice the size of a 4-variable map. We also need another axis. In the absence of holographic projection technology embeddable in textbooks, we need to use a layout like the Fig. 4.10:

With this we can simplify functions like $F = \sum(0, 1, 4, 5, 6, 13, 9, 11, 22, 25, 27)$

Notice in Fig. 4.11 that when a grouping lies entirely on one map the map variable x_4 must be included in the term even though it's not listed on an axis. We can also group cells that are "above" other cells when we consider the vertical orientation of the maps. This takes some getting used to. Add some don't cares into the mix for even more fun.

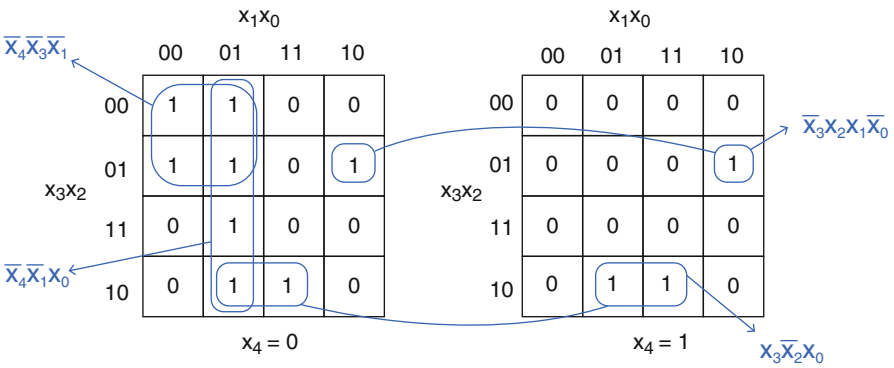


Fig. 4.11 Groupings for five-variable K-map

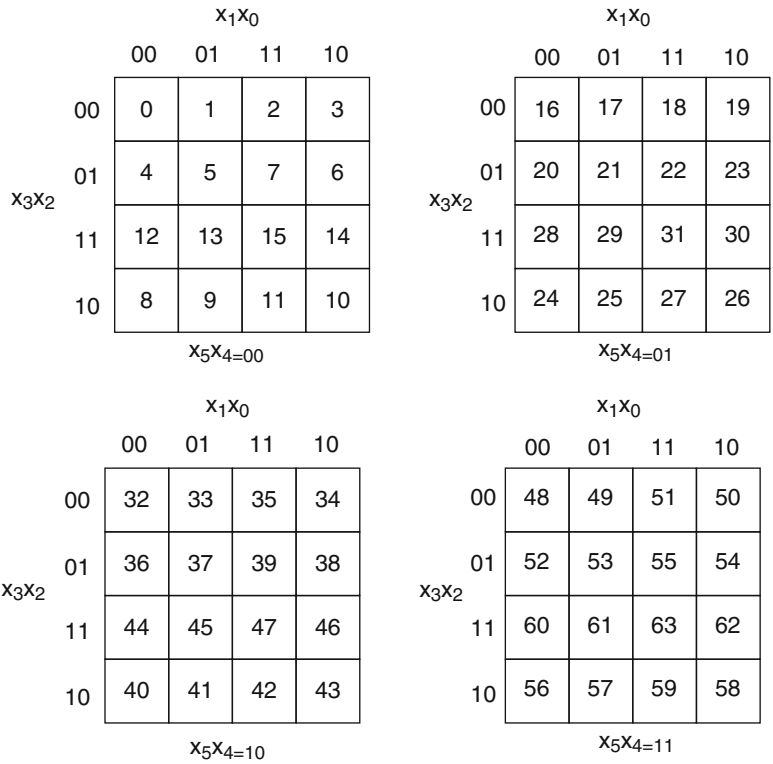


Fig. 4.12 A six-variable K-map

If you dare, check out the six variable map in Fig. 4.12:

This diagram shows the minterm numbers associated with each cell. Notice that the top two maps mirror the 5-variable map but then the bottom-right map is for $x_5x_4 = 11$ so that it contains the highest numbered minterms. This allows us to group through the maps as if they were stacked on top of each other.

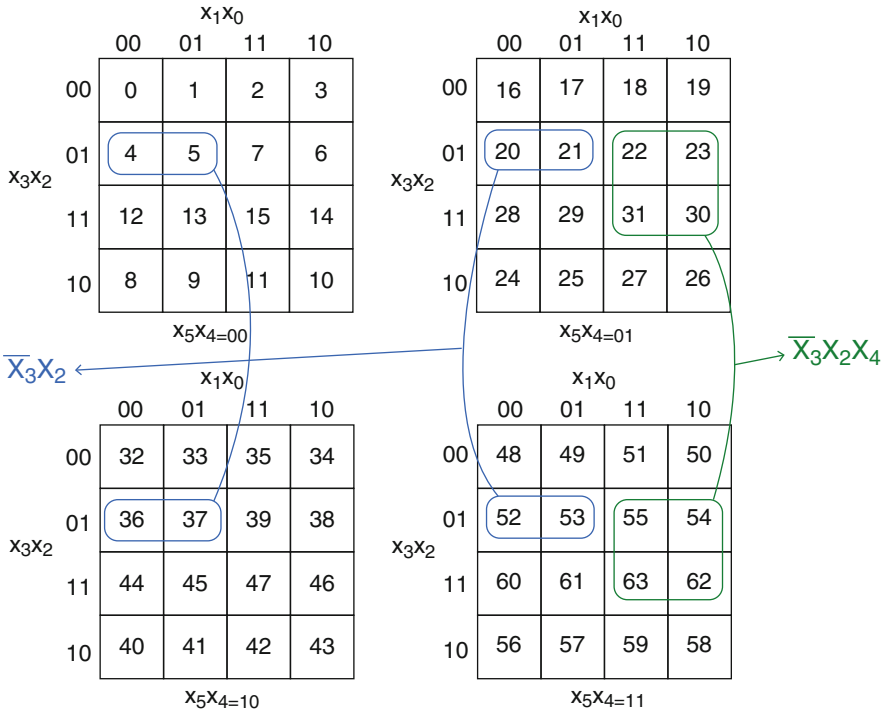


Fig. 4.13 K-map for $F = \sum(4, 5, 20,21,22,23,30,31, 36, 37, 52, 53,54,55,62,63)$:

Consider the function $F = \sum(4, 5, 20,21,22,23,30,31, 36, 37, 52, 53,54,55,62,63)$, whose K-map is seen in Fig. 4.13.

These are interesting mathematically and fall into the category of nigh-unvisulizable concepts called **hypercubes** and are studied in the abstract by topologists. If you have any mathematical inclination, this is a nice reminder that seemingly "useless" mathematics may always be pressed into service to help someone build something cool at some point. So rejoice or beware, depending on your perspective.

Going beyond K-maps of this size on paper is madness, and in the next chapter we'll look at variable-entry maps as a way to avoid the need.

Working with the Product of Sums

The preceding discussion has been entirely based off the SOP form. An analogous development of factoring, grouping, and K-maps can be written for the POS form as well. Let's run through this process using the same function from our prime number detector, $F = \sum(2, 3, 5, 7)$. When working with the SOP form we cared about the

minterms, the 1's in the truth table. When working with the POS form we are going to instead care about the maxterms, the 0's in the truth table. So we write our function as $F = \prod(0, 1, 4, 6)$ and recall the POS canonical form

$$F = (x_2 + x_1 + x_0)(x_2 + x_1 + \overline{x_0})(\overline{x_2} + x_1 + x_0)(\overline{x_2} + \overline{x_1} + x_0)$$

The first thing we did with the SOP form was to factor. Can we factor here? Before we were looking for terms of the form $xy + xz$ that we could change into $x(y + z)$ using the distributive property. We don't see any here, but since this is Boolean Algebra we should not let that deter us. It turns out we have a new distributive property! It turns out that in Boolean Algebra addition distributes over multiplication just like multiplication distributes over addition! (Don't just skim that line—reread it until you get it.) This means we have $x + yz = (x + y)(x + z)$! This is insane for actual real numbers, since claiming $5 + 6^*7 = (5 + 6)(5 + 7)$ leads to retaking algebra class. But this is not algebra on the real numbers, but algebra over our two values logic-1 and logic-0. Things are different here.

Let's apply our new way to factor. We see in the first two terms that $x_1 + x_2$ is added to both x_0 and x_1. This means we can factor out the $x_2 + x_1$ and arrive at

$$F = (x_2 + x_1 + x_0\overline{x_1})(\overline{x_2} + x_1 + x_0)(\overline{x_2} + \overline{x_1} + x_0)$$

which, because $x\bar{x} = 0$, becomes

$$F = (x_2 + x_1)(\overline{x_2} + x_1 + x_0)(\overline{x_2} + \overline{x_1} + x_0)$$

Similarly, the $\overline{x_2} + x_0$ can be factored out of the other term to give us the minimal product of F as

$$F = (x_2 + x_1)(\overline{x_2} + x_0)$$

If we want to use the K-maps to help us see the factoring quickly, we can build the exact same map as we had before and now look at the 0's instead of the 1's (see Fig. 4.14),

We form the terms from the groupings the same way we did the POS form from the truth table: we complement the variables which are 1 and connect variables with a logic OR instead of logic AND. All the same rules for higher dimensional K-maps apply. We interpret don't cares the same way as well, remembering now that when we use the don't care we are making it a 0 instead of a 1.

Fig. 4.14 K-map for $F = \prod(0, 1, 4, 6)$

X_2 \ X_1X_0	00	01	11	10
0	0	0	1	1
1	0	1	1	0

That's it! If you want to know more about minimization, check out the next chapter. If you want to move on to seeing how we implement the logic functions now that we have the minimal form, skip to the chapter after that.

Exercises

4.1 Let $\bar{F} = \prod_{A,B,C}(0,1,5,6,7)$.

(a) Write the canonical Sum Of Products (SOP) form of F.
(b) Use a K-map to find the simplified Product Of Sums (POS) form of F.

4.2 Let $F = \sum_{A,B,C,D}(0,2,5,7,8,12,15) + \sum_{A,B,C,D}X(6,10,13,14)$. Use a K-map to find the minimal SOP form of F.

4.3 Will you always get the same simplified sum form from a k-map that you do from using Boolean algebra? Why or why not?

4.4 Write the logic function $F = AB + \bar{B}\bar{C}$ in canonical SOP and minimal sum forms.

4.5 Write the logic function $F = \prod_{A,B,C}(0,1,3,6)$ in canonical POS and minimal product forms.

4.6 Write $F = \sum_{A,B,C,D}(0,2,5,8,10,13) + X(4,6,7,9,14)$ in a K-map and solve for the minimal sum form.

4.7 The defense systems of a facility incorporate three sensor suites. The sensors cover one of four critical areas during either the day or the night.

Suite A:	Day: areas 1, 2, and 3	Night: areas 2, 3 and 4
Suite B:	Day: areas 2 and 3	Night: areas 1 and 3
Suite C:	Day: areas 1, 2 and 4	Night: areas 1, 2 and 4

Define the following variables:

A = 1 if sensor suite A is working and 0 if sensor suite A is not working
B = 1 if sensor suite B is working and 0 if sensor suite B is not working
C = 1 if sensor suite C is working and 0 if sensor suite C is not working
D = 1 if it is day and 0 if it is night

Define a logic function F to be 1 when each area is covered by at least one working sensor suite and 0 otherwise.

(a) Fill in the truth table for F and find the minimal SOP form of F using a K-map.
(b) How would the minimal logic change if Sensor Suite C always worked? Explain your answer.

4.8 Use Boolean Algebra to reduce $F = \left(\overline{\overline{A+B}}\left(\overline{AC} \bigoplus D\right)\right)$ to minimal SOP form. Also draw up the K-map for this function and find the SOP form from that as well. Your answer should be the same for both methods.

Table 4.2 Truth table for exercise 4.11

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 4.3 Truth table for exercise 4.12

A	B	C	F
0	0	0	1
0	0	1	d
0	1	0	d
0	1	1	d
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

4.9 Design a device that will output a 1 if the input is a perfect square (0, 1, 4, or 9) and a 0 otherwise. Assume the device takes a 4-bit input in unsigned binary format. Your solution should include a completed truth table, K-map, and gate-level logic design diagram.

4.10 Consider F = ∏M(1,2,4,6,8, 11, 13) + ∏ × M(0,7,9,10,15). Write the canonical POS expression and the minimal NOR-NOR expression for F.

4.11 Find the canonical SOP and POS forms for the ogic function given in Table 4.2. Also write the minterm and maxterm decimal representations. Finally, use a K-map to find minimal sum and product forms for F.

4.12 Use a K-map to find minimal sum and product forms for the incompletely specified logic function givein in Table 4.3

4.13 Use a K-map to find minimap POS and minimal SOP forms of the logic function $F(A,B,C,D) = \sum m(1,2,3,5,13) + dc(0,4,7,11)$

4.14 Use a K-map to find a minimal SOP form of the logic function $F(A,B,C,D,E) = \sum m(4,6,7,8,9,11,12,15,16,20,22,23,24,28,30,31)$

4.15 Use a K-map to find the minimal SOP form for the following four-variable logic function: $F = \sum m(0,1,2,4,5,8,15) + X(9,10,11,13,14)$

4.16 Use K-maps to find the minimal sum forms of the following logic functions.

(a) $f(a,b,c) = \sum m(1,3,4,5,6,7)$
(b) $f(a,b,c) = \sum m(0,1,5,6,7)$
(c) $f(a,b,c,d) = \sum m(0,1,2,3,4,5,10,12,13)$
(d) $f(a,b,c,d) = \sum m(0,1,6,7,8,9,13,15)$
(e) $f(a,b,c,d) = \sum m(0,2,3,4,6,7,11,13,15)$

Table 4.4 Truth table for exercise 4.21

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(f) $f(a,b,c) = \prod M(0,2,3,4)$
(g) $f(a,b,c,d) = \prod M(0,1,6,7,8,9,14,15)$

4.17 Use a K-map to find the minimal sum of $f(a,b,c,d) = \sum m(0,1,6,8,13,14) + dc(2,5,7,10)$

4.18 Use a K-map to find the minimal product of $f(a,b,c,d) = \sum m(3,4,5,10,13) + dc(8,11,12,14,15)$

4.19 Find the minimal sum of the following logic function: $f = (\bar{a} + ab)\overline{(b\bar{c} + a)(\bar{a} + c)}$

4.20 Find the NAND implementation of the logic function given by $F(a,b,c,d) = \sum m(1,2,3,6,7,9,10)$.

4.21 For the Boolean function described by the logic table in Table 4.4, give the SOP, POS, minterm, and maxterm forms. Also use a K-map to simplify the function.

4.22 Use K-maps to find the minimal sum and products of the following logic functions:

(a) $f(x,y,z) = \sum m(2,4,5)$
(b) $f(x,y,z) = \sum m(0,1,6) + dc(2,4,5)$
(c) $f(x,y,z) = \prod M(6,7) + dc(0,2,4)$
(d) $f(w,x,y,z) = \sum m(0,3,5,7,15)$
(e) $f(w,x,y,z) = \sum m(0,4,12,13) + dc(2,7,8,14,15)$
(f) $f(w,x,y,z) = \sum m(1,3,5,6,7,10) + dc(4,8,9,11)$
(g) $f(w,x,y,z) = \prod M(0,4,6,10,12,14) + dc(11,13,15)$

4.23 Use K-maps to find the minimal sum forms of the following logic functions:

(a) $f(v,w,x,y,z) = \sum m(0,1,4,5,10,12,13,14,20,21,22,23,28,29,30)$
(b) $f(v,w,x,y,z) = \sum m(0,2,4,6,7,8,10,11,12,13,14,16,18,19,22,29,30)$
(c) $f(u,v,w,x,y,z) = \sum m(1,5,9,13,21,23,29,31,37,45,53,61)$
(d) $f(u,v,w,x,y,z) = \sum m(0,4,8,16,24,32,34,36,37,39,40,48,50,56)$

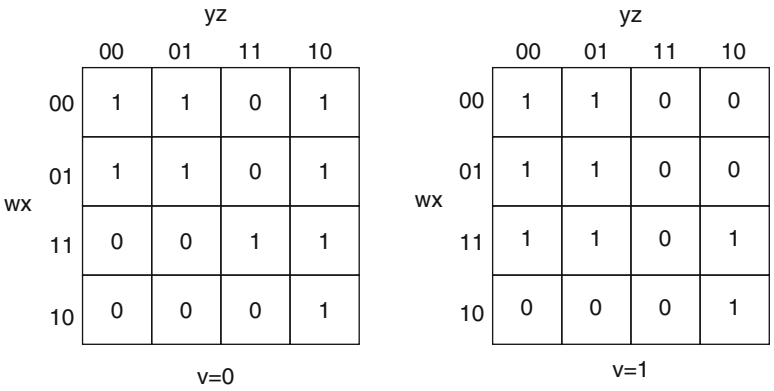


Fig. 4.15 K-map for exercise 4.25

4.24 Suppose the input to a function F is in 4-bit two's complement format.

Define F as follows:

F = x if either adding or subtracting the two's complement number 0011 to the input will cause overflow
F = 1 if adding the two's complement number 0011 to the input results in a positive even number
F = 0 otherwise

Fill in the truth table for F and use a K-map to find the minimal sum form of F. (See Chapter 8 for details on two's complement numbers if the topic is new to you.)

4.25 Find the minimal sum of the logic function described by the five-variable k-map in Fig. 4.15.

4.26 Find the minimal sum of $f(a,b,c,d,e) = \sum m(3,8,9,13,19,23,29,31) + dc(4,5,7,14,20,24,25,27)$

4.27 Find the NAND-NAND implementation of the logic function $F = (\bar{A} + B)(\bar{C} + D)$.

4.28 Find the NOR-NOR implementation of the logic function $F = \bar{A}BC + C\bar{D}$.

Chapter 5
Advanced Logic Function Minimization

While professional digital designers use software tools to synthesize the forms of their logic functions that best fit the design constraints of their projects, studying how the mathematics behind all of this works is still valuable. It's not the sort of thing that leads to a specific skill: no one will care at the technical interview that you're the faster solver of six-variable K-maps in the west. Rather, it's the sort of thing that will enable you to think about things others cannot fathom. The process of understanding—really understanding—Boolean Algebra at a deeper level will empower you in ways you cannot possibly predict.

The material below on variable reduction will be used extensively in the chapters that follow. The rest of the chapter is stand alone, so you can take what you want from it. If what you want is the ability to articulate more clearly, become more knowledgeable about computing, and take the first steps towards contributing to the fundamentals of the field itself, then welcome! We've got an exciting chapter for you!

Variable Entry K-maps

Now we're going to talk about a very useful technique that we'll use throughout the remainder of the text. The idea is that instead of working with a K-map that has a dimension for each variable in the function, we can **transform** the function into another one that is expressed in terms of only some of the variables. In this way we *reduce the dimension*—that is, we use fewer variables than at first glance it appeared we would need—of the necessary K-maps. We call this process **variable reduction**.

Here's a simple example of the basic process. Consider the three-variable function $F = \sum m(2, 5, 6, 7)$. We can reduce it to a two-variable function (Table 5.1).

We convert every two rows of the table into a single value for our new function. Notice that in each grouping of two rows the variables x_2 and x_1 have the same

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_5

Table 5.1 Variable reduction for $F = \sum m(2, 567)$.

X_2	X_1	X_0	F	F(X_0)
0	0	0	0	0
0	0	1	0	
0	1	0	1	$\overline{X_0}$
0	1	1	0	
1	0	0	0	X_0
1	0	1	1	
1	1	0	1	$1 = \overline{X_0} + X_0$
1	1	1	1	

Fig. 5.1 Variable-entry K-map

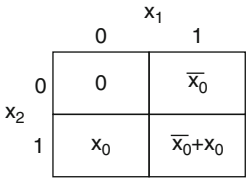
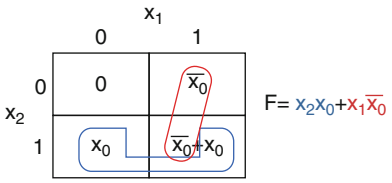


Fig. 5.2 Groupings in variable-entry K-map



values on each line. The variable that changes is x_0 and it is the one being **reduced** or **eliminated**.

In the first grouping we can see that F is equal to 0 regardless of what x_0 is. Therefore, our reduced function is identically 0 throughout that grouping. In the next two lines, however, the function is 1 when $x_0 = 0$ and 0 when $x_0 = 1$ which gives the reduced sum as $1^*\overline{x_0} + 0^*x_0$ which is just $\overline{x_0}$. In the next grouping the situation is reversed and we end up with the sum form of $0^*\overline{x_0} + 1^*x_0 = x_0$. Finally, the last grouping is identically 1 which is going to be useful to write out in the full sum form of $\overline{x_0} + x_0$. You have to get this: go through these details until you're satisfied you have it.

Now, for the K-map. It's called a variable-entry map because we're going to end up putting variables **inside the k-map!**

Figure 5.1 shows this cute little two-variable map, but we can still illustrate the essential characteristics of this procedure. The rules are basically the same: we are after a small number of large groupings, with each grouping limited to a power of two. The new thing we have to deal with is the fact that we have more than two symbols possible in the cells. So instead of grouping "all 1's" or "all 0's" we now must generalize this to "all cells in a grouping must contain the same symbol or symbols." We can see from above that one grouping will containx_0's and another will containx_1's. Oh, and one more thing: we can **break up individual cells at the + sign!** This is demonstrated in Fig. 5.2.

Table 5.2 Variable reduction table for $F = \sum m (0, 4, 5, 7, 10, 12, 13)$

X_3	X_2	X_1	X_0	F	F(X_0)
0	0	0	0	1	$\overline{X_0}$
0	0	0	1	0	
0	0	1	0	0	0
0	0	1	1	0	
0	1	0	0	1	$\overline{X_0} + X$
0	1	0	1	1	
0	1	1	0	0	X_0
0	1	1	1	1	
1	0	0	0	0	0
1	0	0	1	0	
1	0	1	0	1	X_0
1	0	1	1	0	
1	1	0	0	1	$\overline{X_0} + X$
1	1	0	1	1	
1	1	1	0	0	0
1	1	1	1	0	

Fig. 5.3 Variable entry map for $F = \sum m (0, 4, 5, 7, 10, 12, 13)$

x_3 \ x_2x_1	00	01	11	10
0	$\overline{x_0}$	0	x_0	$\overline{x_0}+x_0$
1	0	x_0	0	$\overline{x_0}+x_0$

Wow! Notice that we had to pick off only one of each of the terms from the 1,1 cell to group with the others. This is why it's useful to write 1 as $\bar{x} + x$ whenever we can because it makes it more clear to us that **both** the variable and its complement need to be grouped within that cell.

To synthesize the equation from this map we employ our standard approach of looking at which variables change within the cells of the grouping, but then we also include in the term whatever variable was entered into the map itself. So, we arrive at $F = x_2x_0 + x_1\overline{x_0}$ for the minimal sum. We know we have enough groupings when every variable is part of one. We can't leave even part of the variables within a cell without a grouping.

Let's look at a more complex example using the four-variable function $F = \sum m (0, 4, 5, 7, 10, 12, 13)$ as seen in Table 5.2.

We can work with the reduced function $F(x_0)$ in a three-variable K-map (instead of the four variable map required to work with F) as seen in Fig. 5.3.

The groupings are shown in Fig. 5.4.

Let's talk about the groupings. The 1 term in cell 010 can be broken up to group with the x_0 term in cell 011 and the $\overline{x_0}$ term in cell 000. The other 1 term in cell 110 can group with the entire 1 in cell 010. Remember our rule for variable-entry

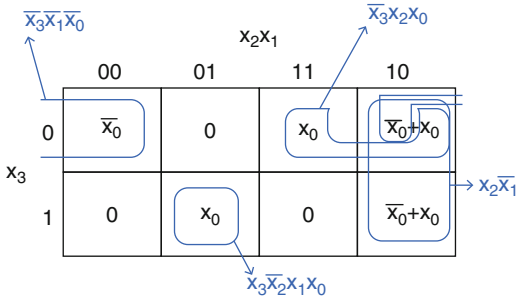


Fig. 5.4 Groupings for $F = \sum m (0, 4, 5, 7, 10, 12, 13)$

Table 5.3 Variable reduction table for $F = \sum(2, 3, 10, 11, 13) + d(6, 7, 8, 12, 15)$

X_3	X_2	X_1	X_0	F	F(X_0)
0	0	0	0	0	0
0	0	0	1	0	
0	0	1	0	1	$\overline{X_0} + X_0$
0	0	1	1	1	
0	1	0	0	0	0
0	1	0	1	0	
0	1	1	0	d	$d\overline{X_0} + dX_0$
0	1	1	1	d	
1	0	0	0	d	$d\overline{X_0}$
1	0	0	1	0	
1	0	1	0	1	$\overline{X_0} + X_0$
1	0	1	1	1	
1	1	0	0	d	$d\overline{X_0} + X_0$
1	1	0	1	1	
1	1	1	0	0	dX_0
1	1	1	1	d	

maps: as long each cell in a grouping contains the same symbols we're good to go. So we can have each cell contain the sum $\overline{x_0} + x_0$. The singleton x_0 in cell 101 can't group with any others and we're left with a full term. Such is life.

We can work with incompletely specified functions in variable entry maps as well. We just need to associate the don't care with the individual terms x_0 or $\overline{x_0}$ instead of with the entire cell. Our notation for this follows from how we compute the sum forms already: we just "multiply" the respective term by the don't care and throw it into the map.

As an example, consider the logic function $F = \sum(2, 3, 10, 11, 13) + d(6, 7, 8, 12, 15)$ whose variable reduction table can be seen in Table 5.3.

If you look at the construction of the don't care terms the exact same way you do the other terms what at first blush appears to be strange formulae such as $d\overline{x_0} + x_0$ starts to make sense. That term has a don't care "multiplied" by the $\overline{x_0}$ term and a 1 "multiplied" by the x_0 term so we get $d\overline{x_0} + x_0$. If the d and 1 were reversed in those two rows we'd have $\overline{x_0} + dx_0$ instead. The two rows that both have d's in them

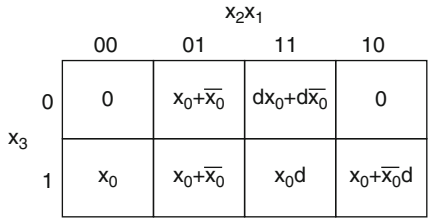


Fig. 5.5 Variable entry K-map for $F = \sum(2, 3, 10, 11, 13) + d(6, 7, 8, 12, 15)$

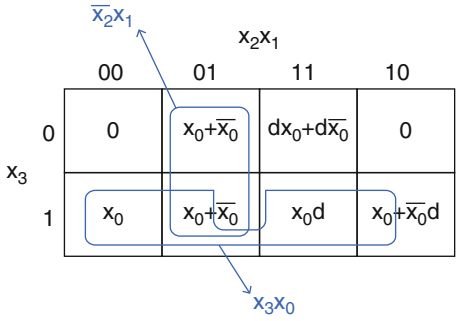


Fig. 5.6 Groupings for $F = \sum(2, 3, 10, 11, 13) + d(6, 7, 8, 12, 15)$

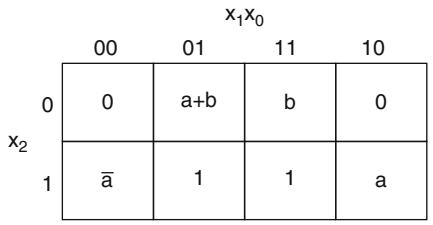


Fig. 5.7 Variable-entry K-map with multiple variables

transform to $d\overline{x_0} + dx_0$for the same reasons of clarity within the map that lead us to write 1 as $\overline{x_0} + x_0$.

Figure 5.5 shows the variable-entry K-map corresponding to the function described in Table 5.3.

In Fig. 5.6 we see the map with the groupings.

Just like with normal don't cares, the ones attached to variables are to be set to 1 or 0 depending on which value best helps to minimize the function. We can see the d in cell 111 should be set to 1 so that we have a grouping of four cells all with x_0 in them along the bottom. We can then group the two $x_0 + \overline{x_0}$ cells together. The other don't cares should be set to 0 because adding groupings with them is unnecessary and would only make the end equation more complicated.

This may seem like a lot of mathematics and a difficult strategy game to play all for reducing a K-map by a single dimension. Well, we can also reduce *multiple variables at once!* This has the potential to leave us with maps like the one seen in Fig. 5.7.

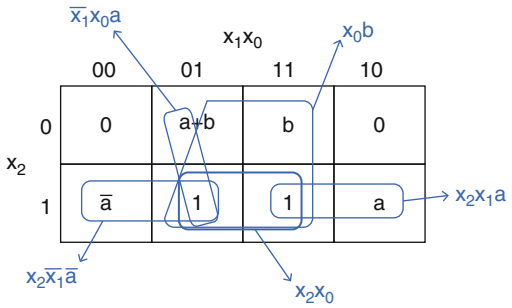


Fig. 5.8 Groupings for multiple variable entry K-map

In this case we've reduced the two variables a and b from the function (Fig. 5.8).

The key to seeing how the groupings in this map work is to remember the rule: **all cells within the same grouping must have the same symbols**. This might mean they all have x_0 or it might mean they all have something like $ab+c$. It all depends on the map.

In this one, we can see that the 1 in cell 101 is grouped with both $\bar{a}$ and a . This is reasonable as that 1 represents the sum $\bar{a}+a$ and we've used this identity before. But we also notice now that we can use the same cell 101 in a larger grouping where each cell is to contain the symbol b. This is also acceptable because $1=\bar{b}+b$ as well. We could write out each possible way to get 1 using the symbols in a map, but once we move beyond a single variable in the map we usually revert to just using a 1. We just have to understand that we can use the 1's to group with any other cell. Just because we've "already" used the cell as representing $\bar{a}+a$ doesn't mean we can't also use it for the b's. All are welcome in the cells of 1.

Look, however, at the other 1 in the map in cell 111. It's grouped with both the b from cell 011 and the a from cell 110. Both of these are fine, but why then does it need to be grouped a *third* time? This is because when you add up all the variables you've grouped in a 1-cell they **must actually sum to 1.** Consider the 1-cell at 101: before being grouped with 111 it's been grouped with a, $\bar{a}$, and b Since $a+\bar{a}+b=1$ everything is good to go and the 1-cell at 110 does not need to group any further with anything else. The 1-cell at 111, however, has only been grouped with a and b. Since $a+b\neq 1$ we still need to group it with something else. We look around and need to add something that will cause it's symbols to sum to 1. If there were an $\bar{a}$ or a $\bar{b}$ still awaiting a group, this cell would be a prime candidate and everything would work out. Since there isn't, it has to group with the 1-cell adjacent to complete the minimal sum.

Yes, this takes some work but in the end it's worth it for those maps where it's appropriate. We'll actually have need to do this later on when we get to large state machine tables that are relatively sparsely populated for some inputs. In these cases, we'll have 1 or 2 cells with some potentially complicated variables entered into them, but the rest of the map will be normal 1's and 0's. With the variable-entry technique we can synthesize what are really 7 or 8 variable functions on a simple 3-variable K-map.

Implicants and Implicates

We speak of the **prime factorization** of a number as the collection of the fundamental units with which we can represent that number. It is a particular *form* that the number can take that is useful for certain applications and which *says something* about the number. The Babylonians used a base-60 number system not because they were too dense to grok the supposed advantages of a base-10 system but rather because 60's prime factorization is *beautiful*: $2 \times 2 \times 3 \times 5 = 60$. With two 2's, a 3, and a 5 as elements of its prime factorization, you can divide 60 into halves, quarters, thirds, or fifths easily, as well as readily attain composites of those: sixths, twelfths, fifteenths, twentieths, and so on. Try that with 10 whose prime factorization is a measly 2 and 5. No thirds or quarters, much less twelfths for base-10. The point is: you can learn a lot about a number from the structure of the primes which compose it.

We have a similar concept for our logic functions. We've already seen that they can be put into many different forms: two normal forms, two (or more) minimal forms, NAND and NOR forms, and many forms in between via the use of the properties and theorems of Boolean Algebra. It turns out that we can identify in our logic functions, like we can in numbers, certain **prime** terms that bear witness to the soul of the function in the same way two 2's and two 5's (boringly) show us the (quite dull) soul of 100.

These terms are called **implicants** for sum forms of functions and **implicates** for product forms. There is a two-valued Boolean Algebra function called *implication* that is used extensively in formal logic and is all but ignored in the implementation of digital logic systems. Its truth table is given in Table 5.4.

That is, *x implies y*, or $x \rightarrow y$, when it's impossible to make $x = 1$ and $y = 0$ at the same time. This follows from our everyday English use of the term *imply*, where the truth of the antecedent x forces the truth of the consequent y. Also, if the antecedent x is false, or logic-0, then we don't much care anything about the truth value of y, so it can be either true or false, logic-1 or logic-0. (When our buddy talks about all the great things he'll do when he wins the lottery we simply nod and say "sure, Vern, sure, whatever," not getting too caught up in the details of his plans because we are confident the antecedent, that he wins the lottery, is now and eternally false.) We see this reflected in the truth table where the function $x \rightarrow y$ evaluates to 1 if $x = 1$ and $y = 0$, or if $x = 0$ and y is anything. It evaluates to 0 only when $x = 1$ and $y = 0$, which corresponds to Vern's failure to live up to his promise to pay you a million dollars of winnings in the event he actually does hit the jackpot.

Table 5.4 Truth table for logical implication

x	y	$x \rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

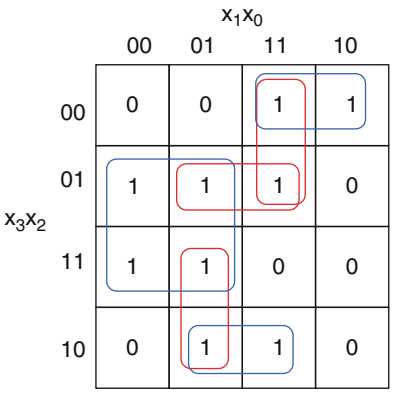


Fig. 5.9 K-map for $F = \sum(2, 3, 4, 5, 7, 9, 11, 12, 13)$

In our digital logic systems, an *implicant* is a term that implies the logic function, that is, *implicant* → *function*. This means two things: (1) when the implicant is 1 the function is 1 and (2) when the function is 0 the implicant is 0.

An *implicate* is the reverse: a term the function implies. That is, *function* → *implicate*. This means two things: (1) when the function is 1 the implicate is 1 and (2) when the implicate is 0 the function is 0.

This is not easy! These are distinct concepts and it takes spending some time with them before it really sinks in what exactly is going on. Don't despair if these sentences read like mental tongue-twisters. Welcome, friend, to the magical domain of formal logic! Luckily, our K-maps help us visualize the underlying mathematical abstraction.

When we work through all the logic we can see that the terms making up the minimal sum form of a function are all implicants. In fact, any term at all that we can group in the K-map is an implicant. Consider the function $F = \sum(2, 3, 4, 5, 7, 9, 11, 12, 13)$ shown in the K-map in Fig. 5.9.

Any grouping at all that we can form, singletons or those *subsumed* by larger groupings (the grouping of cells 0111 and 1100 is *subsumed* by the larger grouping of four cells 0100, 0101, 1100, and 1101, and any singleton cell in this map is also *subsumed* by any number of other groupings) are *implicants* of the function. This is because when those terms are 1 so is the function and for the combinations of inputs that result in the function being 0 those terms are also 0.

Some implicants are special. Some cannot be subsumed by any other implicant. These are the sought-after **prime implicants**. All the implicants in the blue and red groupings are *prime implicants*. None of them fall entirely within the borders of another grouping. The fact that the grouping 1101-1001 is prime may be surprising: it's not part of the minimal sum because it doesn't contain a unique cell, but it is prime because, although it overlaps two other groupings (which are part of the minimal form) it is not entirely subsumed by a single other grouping.

It turns out that in logic functions, unlike in numbers, not all primes are created equal. The groupings in blue in the map are called **essential prime implicants**. They are required terms in every single possible minimal form of the function. They are *essential* to understanding the soul of the function.

The terms in red are non-essential. Of the two near the top, one or the other must be included in the minimal sum form (giving us two possible distinct minimal sums for this function), but the one near the bottom doesn't see inclusion in any minimal sum.

Wow, OK, so what does this all buy us? What does this lesson in logic and vocabulary actually do for us within the realm of digital design? When we name things and categorize things we gain in understanding and are therefore able to think and act with greater precision. In this case, the acting we want to do is the creation of minimal forms. Remember that for the simple low-variable functions we can use the K-map method but for industrial scale problems it's just not practical to do all this minimization work by hand. We want to **write programs to find the minimal sums for us**. We can now formulate the algorithm we need to implement the minimization of a logic function as a **search** for the *prime implicants***,** and a noting of the *essential prime implicants*, of a logic function. We illustrate one of these algorithms in the next section.

We can run through all the same logic using **implicates** as well and couch the quest for a minimal product form as that of finding the essential prime implicates. Typically in digital design we're content with the sum form, but for functions whose product form is much more efficient to implement then such a process would be in order.

The Quine-McCluskey Algorithm

At the end of the day, automated logic function minimization techniques rule the roost when it comes to professional digital design tasks. This section looks into one such approach called the Quine-McCluskey Algorithm. It's a starting point for your own investigations into coding up a routine that can do this work.

This algorithm is conceived as one that searches the space of a function's implicants to find the prime implicants and, out of these, identifies which are essential. As in many of our digital design algorithms, our first pass is to consider what we as humans do and then try to have the computer follow our lead. In this case, however, it's hard to say to the network of logic gates "look at this K-map and circle the adjacent 1's." So, we have to think a bit harder and recall exactly where the K-map originated and what, algebraically, is happening when we play our grouping cells strategy games.

It's all about finding the one-bit differences between minterms and factoring out variables until we are left with nothing left to reduce. We then have our minimal sum. The Quine-McCluskey algorithm follows that path in a way a computer can

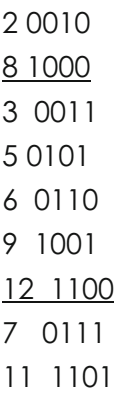


Fig. 5.10 Step one of Quine-McCluskey for $F = \sum(2, 3, 5, 6, 7, 8, 9, 11, 12)$

understand rather than in the visual manner of the K-map that makes the process easier for humans.

The algorithm consists of four steps:

1. Sort the minterms into categories based on the number of 1's in them
2. Analyze adjacent categories for 1-bit differences and reduce
3. Repeat Step 2 until no more reductions are possible
4. Of all the implicants remaining, the ones not used to create a reduced term are the prime implicants

Once the prime implicants are discovered, we can then suss out the essential prime implicants using several methods. We'll talk about a tabular method that works well enough for demonstrating the power of the Quine-McCluskey algorithm. If you do go to implement this and want to see how the commercial software packages that perform this sort of optimization operate, then you may encounter more computationally efficient methods. But for our purposes, to illustrate a single algorithmic approach, it will work just fine.

Onward, to an example!

Consider the function $F = \sum(2, 3, 5, 6, 7, 8, 9, 11, 12)$. For step 1, we arrange the minterms not by magnitude but by the number of 1's they contain (see Fig. 5.10).

For step 2, we compare each minterm with all the minterms in the category previous, and create reduced forms with blanks in the positions corresponding to one-bit differences. This indicates a term has been eliminated from the equation.

As seen in Fig. 5.11, the notation (2,3) means we've formed this term by grouping minterms 2 and 3. The fact that we search for groupings only by checking the category previous to the current minterm's category ensures we don't have to worry about our algorithm doubling up and also, say, creating a (3,2) term.

For step 3, in Fig. 5.12, we continue this process. Now when we check for our one-bit differences we must ensure the blanks align first and only compare those terms for which that's the case.

(2,3) 001_
(2,6) 0_10
(8,9) 100_
(8,12) 1_00
(3,7) 0_11
(5,7) 01_1
(6,7) 011_
(9,11) 10_1
(3,11) _011

Fig. 5.11 Step two of Quine-McCluskey for $F = \sum(2, 3, 5, 6, 7, 8, 9, 11, 12)$

(2,3,6,7) 0_1_

Fig. 5.12 Step three of Quine-McCluskey for $F = \sum(2, 3, 5, 6, 7, 8, 9, 11, 12)$

	m2	m3	m5	m6	m7	m8	m9	m11	m12
(8,9)						x	x		
(8,12)						**x**			**x**
(5,7)			**x**		**x**				
(3,11)		x						x	
(9,11)							x	x	
(2,3,6,7)	**x**	**x**		**x**	**x**				

Fig. 5.13 Quine-McCluskey Implicant table for $F = \sum(2, 3, 5, 6, 7, 8, 9, 11, 12)$

We notice in this example that we can combine the terms (3,7) and (2,3) as well as the terms (6,7) and (2,6). Well, both of these give us the term (2,3,6,7) so in reality there is only a single grouping left for Step 3 to chronicle.

Step 4 says that we can identify the prime implicants as the ones not used in any reduction. In our example, that leaves terms (8,9), (8,12), (5,7), (9,11), (3,11), and (2,3,6,7) as our prime implicants.

To find a minimal form we now need to array the minterms making up the function against the prime implicants available to us. The tabular methods compares them in table form as demonstrated in Fig. 5.13:

The essential prime implicants are found by finding the columns that contain exactly one x in them. We must have the corresponding term in the minimal sum, so we highlight all minterms covered by this term. So, (2,3,6,7) is required for m2 and m6, (5,7) is required for m5, and (8,12) is required for m12. After we establish these essential prime implicants, we look and see what minterms are still in need of covering. In this case we still need to cover m9 and m11. What is the fewest number of additional prime implicants we would need to cover these? We could choose (8,9) and (3,11), but it's more efficient to simply choose the (9,11) term we have

available. So that becomes a non-essential prime implicant that is present in the minimal form and we construct the equation:

$F = x_3\overline{x_1 x_0} + \overline{x_3}x_2x_0 + x_3\overline{x_2}x_0 + \overline{x_3}x_1$

We generate the terms in the equation from the expressions of the terms we use from the table. For example first term corresponds to (8,12) whose representation in our algorithm is 1_00. This means the most significant variable, x_3, is present and the two least significant variables x_1 and x_0 are present in their NOT forms. The variable corresponding to the _ in our representation, x_2, has been factored out of the term and does not appear in the equation.

And that's it! That's the Quine-McCluskey algorithm. While there are any number of effiiciencies and algorithmic speedups and tweaks that a computer scientist can apply to this approach, the core algorithm is a demonstration of how we can go from requiring humans staring at K-maps (or doing algebra really fast) as our only method of finding minimal forms to sitting back and letting our software help.

Exercises

5.1 Let F be defined by the truth table in Table 5.5:

Use the Quine-McClusky algorithm to find the minimal sum form of F.

5.2 Let F be defined by the truth table in Table 5.6.

Use a variable-entry K-map to find the minimal SOP form of F.

5.3 Consider the logic function $F = \sum_{A,B,C,D}(1, 3, 4, 9, 11, 12, 15)$. Find the minimal sum of products form using the Quine-McClusky algorithm (be sure to clearly state the prime implicants.) Compare this result to the minimal form you get from a K-map.

5.4 Use the Quine-McCluskey algorithm to find the minimal SOP form of the three variabl logic function given by $F = \sum m(1, 3, 4, 6, 7)$

Table 5.5 Truth table for exercise 5.1

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 5.6 Truth table for exercise 5.2

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	x
0	1	0	0	x
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	x
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Table 5.7 Truth table for exercise 5.5

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
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	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

5.5 Use a variable-entry K-map to calculate the minimal sum form for the logic function given in Table 5.7.

5.6 Use a variable-entry K-map to find the minimal SOP form for the logic function given in Table 5.8:

Table 5.8 Truth table for exercise 5.6

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

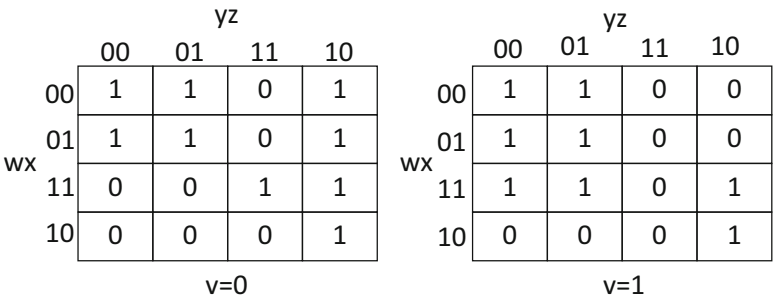


Fig. 5.14 K-maps for exercise 5.9

5.7 Use the Quine-McCluskey algorithm to find a minimal SOP form of the following three variable logic function: $F = \sum m(0, 1, 4, 5, 7)$

5.8 Show how to implement the function $f(a, b, c, d) = a + b\bar{c} + \bar{b}d$ using only NAND gates.

5.9 Find the minimal sum of the logic function described by the five-variable k-map in Fig. 5.14.

5.10 Use variable reduction to find the minimal sum of four variable logic function given in in Table 5.9 using a three variable K-map.

Table 5.9 Truth table for exercise 5.10

a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

5.11 Find the minimal sums for the logic functions in Fig. 5.15 already in K-map form:

5.12 Use a K-map to show all implicants of the following logic functions. Indicate which implicants are prime and essential.

(a) $f(w,x,y,z) = \sum m(0,3,7,9,12,13,14,15)$
(b) $f(w,x,y,z) = \sum m(1,2,3,5,7,11,12,13,15)$

5.13 Use a 3-variable k-map to find the minimal sum of this 4-variable function (Table 5.10).

5.14 For the logic function indicated by the k-map in Fig. 5.16, circle the prime implicants.

Are any of the prime implicants non-essential? If so, indicate which one(s).

5.15 Find the simplest form of the function f(a,b,c,d) which is described by the K-map in Fig. 5.17.

5.16 Circle all the prime implicants in the K-map of Fig. 5.18

Fig. 5.15 K-maps for exercise 5.11

a

x \ yz	00	01	11	10
0	u	1	0	0
1	0	w	0	0

b

x \ yz	00	01	11	10
0	0	u+w	u	0
1	0	ū+w	1	w

c

x \ yz	00	01	11	10
0	0	1	w	0
1	$\underline{0}$	v	u+w	1

d

A \ BC	00	01	11	10
0	u	0	0	u
1	ū	1	w	w

e

x \ yz	00	01	11	10
0	u	1	0	0
1	v	v+w	0	1

Table 5.10 Truth table for exercise 5.13

w	x	y	z	f	f(z)
0	0	0	0	0	
0	0	0	1	1	
0	0	1	0	1	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	x	
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	x	
1	0	1	1	1	
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	1	
1	1	1	1	0	

Fig. 5.16 K-map for exercise 5.14

wx \ yz	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	1	1	0	1
10	0	0	0	0

Fig. 5.17 K-map for exercise 5.16

d \ ab\c	11	01	00	10
00	d	1	0	d
01	0	1	0	0
11	0	0	0	0
10	1	1	0	1

Fig. 5.18 K-map for exercise 5.17

AB \ CD	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	0	1	0	0
10	0	1	0	1

Chapter 6
Logic Gates

Armed with our minimal forms we feel mighty indeed, and we are ready to build the device we've designed. We need to physically perform the operations demanded by the equations derived for our logic functions. If our minimal form is $F = x_2\overline{x_1} + x_2x_0$, say, then we need to perform one NOT operation, one two-input OR operation, and two two-input AND operations. We can draw a schematic of the circuit layout required, as seen in Fig. 6.1:

We can see here three different symbols representing our **logic gates** which physically perform the required calculations. The lines indicate actual physical wires carrying the electronic signals we label with our variables x_2 , x_1, and x_3.

Figure 6.2 presents a summary of the basic gates:

We should be able to translate from the circuit diagrams for logic gates to the Boolean Algebraic equations and vice versa. For example, if we have the circuit given in Fig. 6.3

we should be able to write the function $F = x_2x_1 + \overline{x_2}x_0$.

Sometimes we write the inversion on the input to the gate instead of writing out the full NOT gate. So, the two circuits presented in Fig. 6.4 would be equivalent.

The circle is called an **inversion bubble** and is used throughout our digital design diagrams on either the input or the output of a gate to indicate the NOT operation.

Programmable Logic Arrays

While we can ideally implement our logic function's minimal form using only the logic gates we need, sometimes such fabrication is overkill. It's not always critical to a given project's needs to absolutely optimize the area of a circuit design. For these reasons, special arrangements of gates are available which help implement functions in their canonical forms rather than in their minimal forms. These are **called programmable logic arrays**, or PLA's.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_6

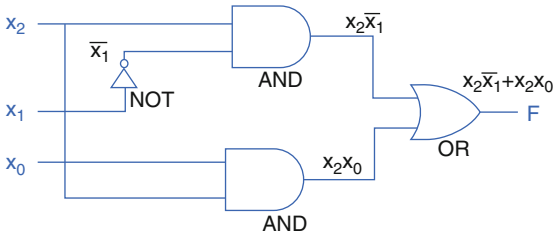


Fig. 6.1 Logic gate diagram for $F = x_2\overline{x_1} + x_2x_0$

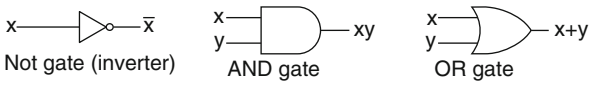


Fig. 6.2 Basic gate schematics

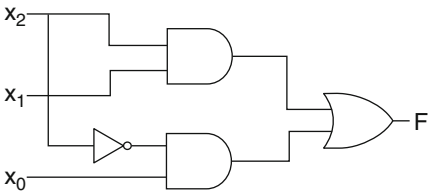


Fig. 6.3 Logic gate diagram for $F = x_2x_1 + \overline{x_2}x_0$

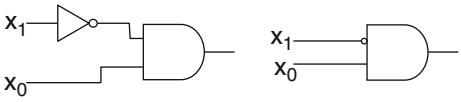


Fig. 6.4 Logic gate equivalencies with inversion bubble

PLA's typically can be found in AND-OR forms and OR-AND forms. See Fig. 6.5.

To use one of these PLA's, the designer must connect each of the six possible inputs (the three variables and their complements) to each gate of the AND or OR array. This structure permits any function to be quickly and readily implemented in its canonical form. The AND-OR array is a good choice for the SOP form and the OR-AND array works well for the POS form. These arrays are yet another reason why in some cases we prefer the canonical forms over the minimal forms. We're not always fabricating an entire circuit from scratch. Sometimes we are building a low-cost prototype or adding a component to a larger system or testing what works and what doesn't. It's critical to understand that these different forms of functions and the varying implementation technologies all have their place. Not every design project operates under the same constraints and figuring out what yours are can go a long way towards engineering the best design for your current metrics.

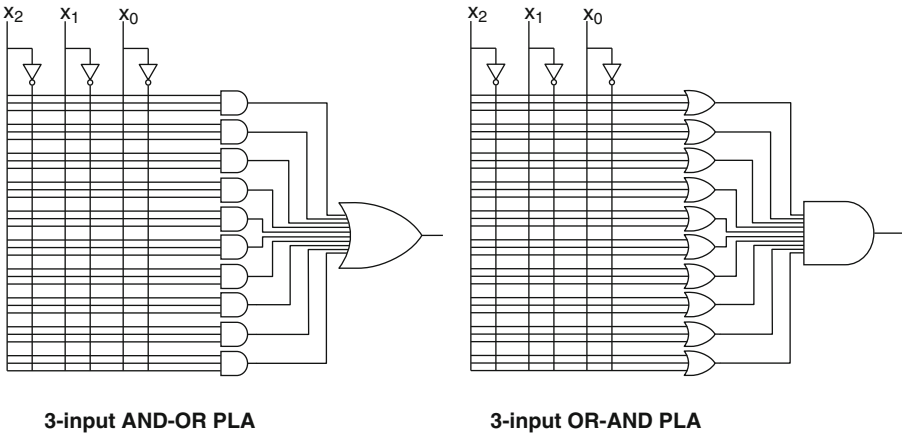


Fig. 6.5 Programmable Logic Array (PLA) diagrams

x	y	XOR	NAND	NOR
0	0	0	1	1
0	1	1	1	0
1	0	1	1	0
1	1	0	0	0

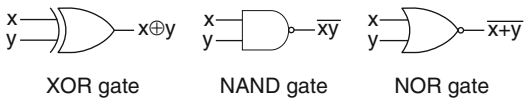


Fig. 6.6 XOR, NAND, and NOR logic gates

XOR, NAND, and NOR

There are some other gates which are helpful as well. Their truth tables are specified and their symbols given in Fig. 6.6.

The XOR gate, called the **exclusive or**, symbolized by the operator $\oplus$, is the same as OR except for the case when both inputs are 1, then it returns a 0. It *excludes* the case where both inputs are 1, while our standard OR, which we can call the *inclusive or*, returns a 1 for that case. XOR has a lot of applications and we'll keep it in mind as a useful gate from here on out. One of it's features is that it can be a *parity detector*, as it is 1 when there are an *odd* number of 1 inputs and 0 when there is an *even* number of 1 inputs (yes, this is a valuable thing to detect for some applications.)

The NAND and NOR gates are the basic AND and OR gates with an inverter tacked on. Why is that helpful? You can think of them as *active-low* versions of AND and OR. Whereas AND will return 1 only when all inputs are 1, NAND returns 0 only when all inputs are 1. Whereas OR will return 1 when at least one input is 1, NOR returns 0 when at least one input is 1. These features themselves are useful, but there is one other reason NAND and NOR gates are powerful: they act as **universal gates**.

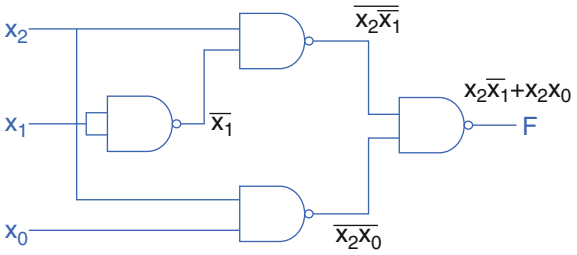


Fig. 6.7 NAND gate implementation of $F = x_2\overline{x_1} + x_2x_0$

What happens when you put an input x through *both* inputs of the NAND gate? You get $\overline{xx}$ which is just $\bar{x}$. So the NAND gate can work like a NOT gate. What if you put x and y into a NAND gate and then feed that result back into another NAND gate? The first one will give $\overline{xy}$ and the next will invert that to remove the negation and leave xy. So, with two NAND gates we can perform the AND operation. Finally, if we use NAND gates to invert both x and y *before* sending them through a third NAND gate we get $\overline{\bar{x}\bar{y}}$ which is just $x+y$ (via DeMorgan's law from the previous chapter.) So, with three NAND gates we can perform the OR operation. This means with *only NAND gates* we can implement **any** logic function! All we need is NAND! (Or NOR, as you can do the same thing with that gate.)

As an example, take our function $F = x_2\overline{x_1} + x_2x_0$ from above. We can implement it with four NAND gates (see Fig. 6.7 for the diagram.)

If you skimmed over the advanted Boolean Algebra section, following the math (which depends on DeMorgan's Law) might be a bit much, but it's worth working through one time. Make sure you understand how this diagram is successfully implementing the function F and why it's equivalent to the previous implementation with AND, OR, and NOT gates.

If you can stand some algebra, here's how you can find the NAND logic form of a function in minimal sum form. You negate it twice and use DeMorgan's Law one time on the innermost negation:

$$F = x_2\overline{x_1} + x_2x_0 = \overline{\overline{x_2\overline{x_1} + x_2x_0}} = \overline{\overline{x_2\overline{x_1}}\,\overline{x_2x_0}}$$

Now everything is a NAND operation.

You can run through a similar process to write minimal products as NOR logic. If you want to convert, say from a minimal sum into a NOR form it's best to go back to the truth table or K-map and derive anew the other minimal form and then convert. Synthesis tools can do this for you in software, but it's good to have a feel for how the algebra works out. The idea that NAND gates can implement any function is so powerful and important that it's worth going through the background justification at least once.

Bitwise Logic

One of the reasons we're encouraging you so heavily to break away from the idea that logic-1 means "true" and logic-0 means "false" is that it's incredibly helpful when doing digital design to conceptualize the logic gates in other ways. For example, we can think of a NAND gate as the gate that detects (evaluates to logic-1) whether there is a 0 in the input and an OR gate as the one that detects (evaluates to logic-1) whether there is a 1 in the input. These ideas have nothing to do with any notion of true or false.

In the same vein, we can consider that if we OR a value with 1 we can force the value to become 1 (we call this *setting the bit*.) Similarly we can see that if we AND any value with 0 we force the value to become 0 (we call this *clearing the bit*.) Operations on individual bits are so common in computing that we have a special name for them: **bitwise operations**.

Even our XOR gate is useful in bitwise operations. Since we can think of XOR as a parity detector (it's 1 when there are an odd number of 1's in the input and 0 when there are an even number of 1's in the input) removed from any truth or logic connotation tied up in its name "exclusive or" we can see how it can help in our task. XORing with a 1 will always change the parity, because adding 1 to an odd number gives an even number and adding 1 to an even number gives an odd number. Therefore, we can use XOR gates with 1 to complement individual bits in our subtraction operation. Check out the truth table for XOR if you need convincing, but please be aware that how you think about XOR matters. It's one thing to think of it as one interpretation of how we use "or" in English, but while that's helpful when working in formal logic and predicate calculus, it's of limited utility when it's time to incorporate XOR gates in digital designs.

Looking at our logic gates from a pure numeric bitwise perspective is invaluable. Each of the main operations is highlighted in Fig. 6.8.

For example, if we have the number x=10110 and we want to clear the x_2 and x_1 bit positions we can use the bitwise operation x *AND* 11001 which results in 10000. The value 11001 is called the **mask** as it determines what of the original value is preserved and what of it is changed. In this case the result is that we get a 0 everywhere the mask is 0 and the original bit string remains unchanged everywhere the mask is 1.

If we instead use a mask of 00001 and the OR operation we end up setting the x_0 bit. If we use a mask of 11000 and the XOR operation we end up complementing the x_4 and x_3 bits.

We're going to be representing a lot of different types of information with bit strings and accessing individual bits is something we'll care about doing. It is also a

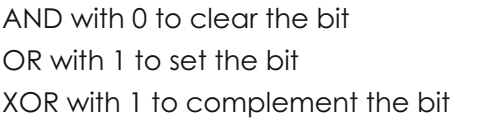


Fig. 6.8 Uses for the bitwise logic operations

great way to get a better digital logic design feel for what the so-called "logic" operations do. Often we stay too tied to the origins of these terms in actual propositional logic and we fail to see the true breadth of their engineering application.

CMOS

One final step remains: what, exactly *is* a gate? So far a gate is just a symbolic representation of one of our logic operations on a circuit diagram. Our goal is to find a physical realization of our equations, so we need now to figure out how to physically implement the logic gates. Such a physical implementation requires us to output the correct physical signal for each input as described in the gate's truth table.

Logic gates can be built in many ways. In mechanical computers they can be constructed by gear systems. In molecular computers they can be represented chemically. You can even use Lego toys to build (as crude and as unworkable as they are awesome) gates. Dominoes even make for a fun gate implementation (albeit one that can only be run once.) But, our focus here is on the *electronic computer* so we are going to focus on electronic means of implementing gates.

In our first chapter we discussed the metal oxide semiconductor field effect transistor, or MOSFET, and its circuit diagrams. It's a good time to review that section, because now we're going to show how to take that basic electronic component and use it to build physical logic gates.

Recall the nFET and pFET seen in Fig. 6.9.

We can now recognize that the nFET is an **active high** device and the pFET is an **active low** device. This means that in our logic implementations we will hook complemented variables up to the pFETs and noncomplemented variables to the nFETS.

Remember also that we use the nFETs to pass a **strong 0** and the pFET to pass a **strong 1**.

As far as logic design goes, this means it's great to use nFETs when we have 0's and it's great to use pFETs when we have 1's. Notice the inversion bubble on the

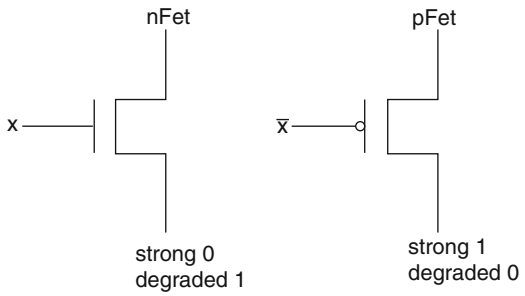


Fig. 6.9 nFet and pFet schematic

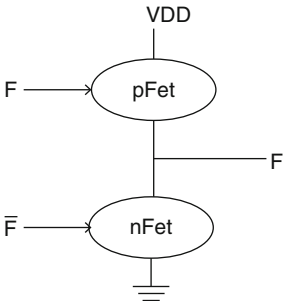


Fig. 6.10 CMOS transistor array framework

pFET diagram. This means that the pFET is "on", that is, it permits its signal to be transmitted, when the gate voltage is low. In contrast, the nFET is "on" and permits its signal to be transmitted when the gate voltage is high.

Since our logic functions sometimes output a 0 and sometimes a 1, we want to use both types of MOSFETS together in our design.

The best transistor layout we have right now, for most purposes, does just that and uses each type of MOSFET for the purpose it excels at. If we want to implement a logic function we can use an nFET array to transmit low values of the function and we can use a pFET array to transmit high values. The diagram for this type of circuit is shown in Fig. 6.10.

We call this use of nFETs and pFETS **complementary**, and we call this array layout **CMOS.** It is the dominant implementation technology in use today.

The key to this implementation is that when $F = 1$ the path through the pFETS from VDD to F is open while the path through the nFETS from F to ground is closed. Therefore, the strong 1, the high voltage in the system, gets passed through the pFETS to F. When $F = 0$ we have the reverse: the path from VDD to F through the pFETS is closed while the path from F to ground through the nFETS is open. Therefore, the strong 0, the low voltage in the system, gets passed through the nFETS to F. We don't want a situation where both the pFET and nFET arrays are open at the same time!

Let's see how to build an AND gate out of CMOS transistor arrays.

First, we need to compute both the logic function and its complement. In this case, we have

$$F = xy$$
$$\bar{F} = \bar{x} + \bar{y}$$

We want to implement F using the pFET array. This means we want the path from VDD to F open when variables x and y are logic-1. To implement this, we arrange the transistors in **series** to indicate both variables must be 1 in order for the path to be open. Also, we have to remember that the pFETS are active-low so we will need to activate them with $\bar{x}$ and $\bar{y}$ instead of with x and y.

Figure 6.11 shows the pFET array for our logic function F.

The nFET array requires implementation of $\bar{F}$. In this case, we need the connection through the nFETS, from F to ground, to be open when $F = 0$, which is when $\bar{F} = 1$. Our equations indicate this happens when either $\bar{x}$ or $\bar{y}$ is logic-1. To implement this, we arrange the transistors in **parallel** to indicate either variable may be 1 in order for the path to be open.

Figure 6.12 shows the nFET array for our logic function F.

Taken together, the full CMOS transistor array for the function $F = xy$ is shown in Fig. 6.13.

We still need to invert both x and y. The CMOS inverter is straightforward (Fig. 6.14).

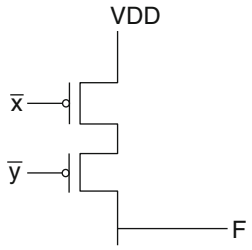


Fig. 6.11 pFet array for $F = xy$

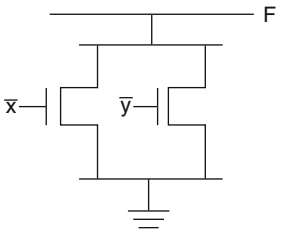


Fig. 6.12 nFet array for $F = xy$

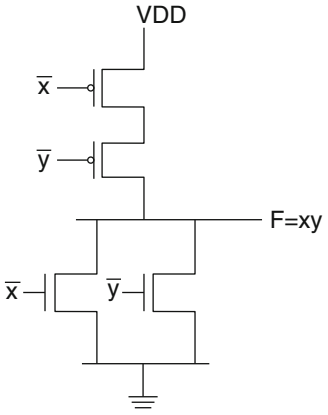


Fig. 6.13 CMOS transistor array for $F = xy$

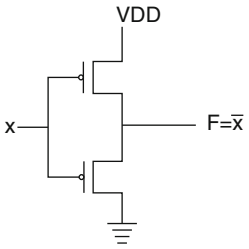


Fig. 6.14 CMOS inverter

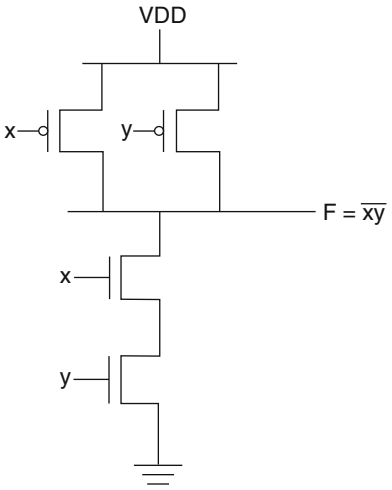


Fig. 6.15 CMOS array for the NAND logic function

Make sure you understand how this inverter is built using the principles laid out in the preceeding section.

We see the CMOS inverter requires two transistors. Our AND gate, then, requires four transistors in the main array plus two inverters for a total of 8 transistors. What about NAND? In that case we have $F = \overline{xy} = \bar{x} + \bar{y}$ and $\bar{F} = xy$. We end up with the CMOS array given in Fig. 6.15.

This does not require any inverters, so we have a total of four transistors required to implement a NAND gate. From this we can see that not all gates are created equal! An AND gate is twice as expensive as a NAND gate (in these implementations.) This is something to consider when deciding on what form a function should take. The NAND form is often less expensive to implement in CMOS.

Electronic Properties of Gates

Now that we see the electronics underlying our logic functions and gates, we can discuss a few of their physical properties. Perhaps the most important of these for our concern is the concept of delay. Since we now can see our gates as physical, we

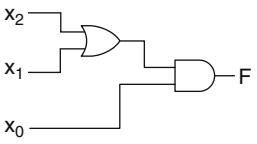


Fig. 6.16 Two-level logic diagram

have some general idea that the number of gates a signal must pass through has an effect on the actual time it takes for that transmission to occur. We care about this because when we are fitting digital devices together in a larger system, we need to make sure their timing lines up. We don't want one device sending inputs to another faster than it's capable of processing them.

We define two types of delay, **propagation** delay and **contamination** delay. Propagation delay measures the time it takes for the output to stabilize in response to a change in inputs. We determine this by considering the time required for a signal to move through the **critical path** in a circuit—that is, the longest possible delay from input to output. Contamination delay refers to the time it takes not for the output to finalize but simply for the output to begin to change in response to a change in inputs. It's measured via the shortest path through a circuit. For example, in the circuit given by the diagram in Fig. 6.16

we have a propagation delay of two gates and a contamination delay of just one. As soon as x_0 changes, our logic function can potentially change, even before the OR gate has a chance to process changes in x_2 and x_1. For example, if we have $x_2 = 0$, $x_1 = 1$, and $x_0 = 0$ so that $F = 0$ and then the inputs change to $x_2 = 0$, $x_1 = 0$, and $x_0 = 1$, we'll have a brief window where $F = 1$ before finally settling down to $F = 0$ again. This is because the contamination delay changes the output before the propagation delay finalizes it.

Exercises

6.1 Implement an XNOR (exclusive-NOR) gate using only NAND gates. That is, draw the gate-level logic diagram for an XNOR function using only NAND gates.

6.2 Let $F = A \bigoplus B + A\bar{C}$.

(a) Draw the gate-level logic diagram for F (using any gates you want to use).
(b) Fill in the truth table for F.

6.3 Redraw the logic diagram given in Fig. 6.17 in simplest possible form. (Hint: write the diagram as an equation and use DeMorgan's Law and the axioms of Boolean algebra to simplify it).

6.4 Consider the logic function $F = A\bar{B}\bar{C} + A\bar{B}C + ABC + \bar{A}BC$. Implement this function using CMOS transistor logic.

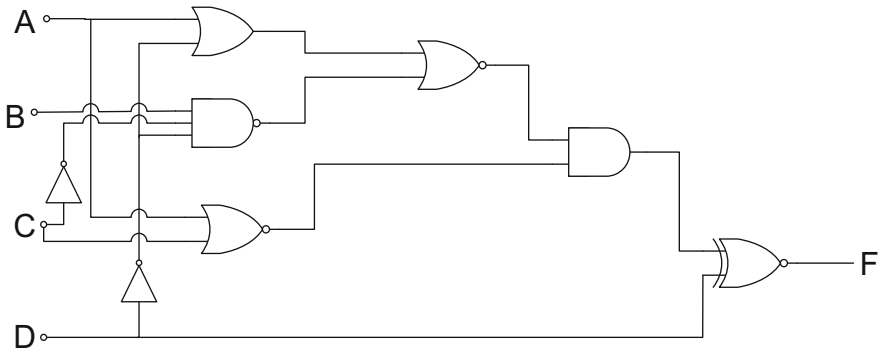


Fig. 6.17 Logic gate diagram for exercise 6.3

6.5 Draw a logic diagram using only NOR and NOT gates that will perform the 3-input AND function.

6.6 For the following logic functions find the truth table, the canonical SOP and POS forms, and draw the gate-level implementation of these forms.

(a) $F = \bar{A}B + A\bar{C} + BC$
(b) $F = A\bar{B}\bar{C} + \bar{A}B + C$

6.7 Implement the logic function $F = AB\bar{C} + \bar{A}BC$ using CMOS transistors. Go through all the stages of CMOS logic function design (identify SOP and POS forms for both F and F-inverse, choose which equation for each to use in the implementation, figure out whether an output inverter is necessary, and draw the final implementation diagram.)

6.8 Consider a logic function that takes input in XS3 format and outputs a 1 when the input is prime (i.e., 2, 3, 5, 7). Implement this function using CMOS transistors.

6.9 Draw the gate-level implementation of the following logic function: $F = \bar{A} + B(A + \bar{C})$

6.10 Implement the following logic function using only NAND gates: $F = \overline{AB} + C(A \bigoplus B)$

6.11 Draw a CMOS circuit to implement the function $F = A\bar{B} + BC + A\bar{C}$.

6.12 Draw a CMOS circuit to implement the function $F = A\left(\overline{\bar{B} + C}\right)$.

6.13 Implement the XOR function using CMOS transistor arrays.

6.14 Draw the gate-level implementation of the following logic function: $F = A\bar{B} + A(B + \bar{C})$

6.15 Find the minimal NAND gate realization of the logic function $F = \bar{A}B + \bar{B}C$. Show the algebraic formulation as well as the gate-level implementation.

6.16 Draw the gate-level implementations of the following logic functions. Use only AND and OR gates and inverters. Do not simplify the algebra; just implement the functions in the form given.

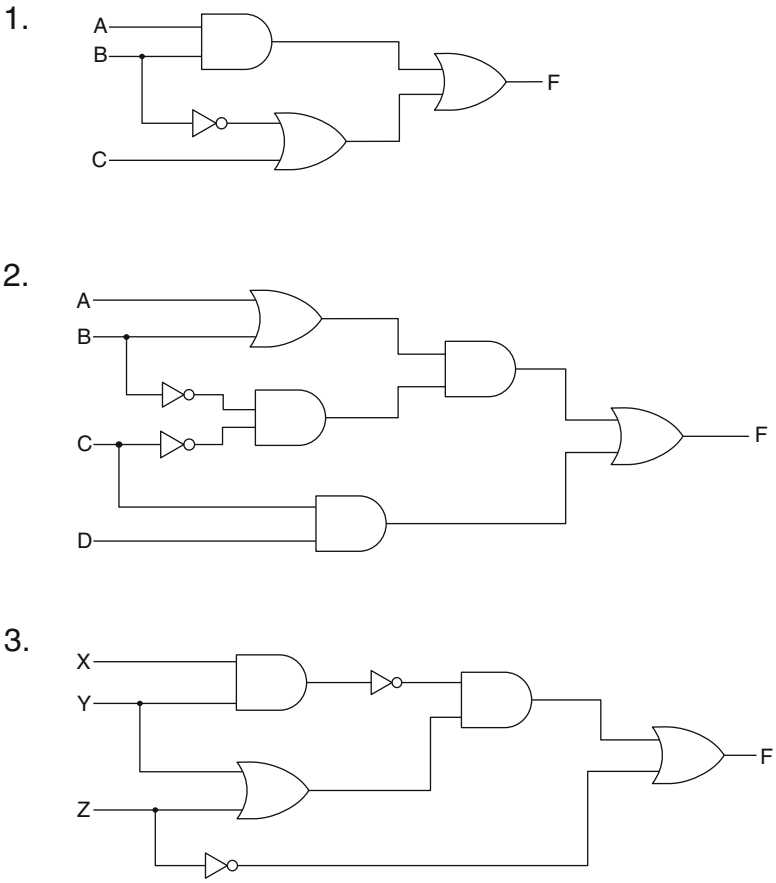


Fig. 6.18 Logic diagrams for exercise 6.17

(a) $f(x, yz) = xy + \bar{x}z + x(y + z)$
(b) $f(w, x, y, z) = x(w + \bar{y}(z + \bar{x}))$
(c) $(a, b, c) = a\bar{b}c + ab\bar{c} + \overline{abc}$

6.17 Give the Boolean equation corresponding to the following gate-level circuits in Fig. 6.18.

6.18 Implement each of the following functions on an AND-OR PLA:

(a) $F = AB\bar{C} + A\bar{B}\bar{C} + \bar{A}BC + \bar{A}\bar{B}C + ABC$
(b) $F = (A + \bar{B} + C)(\bar{A} + B + C)(A + \bar{B} + \bar{C})$

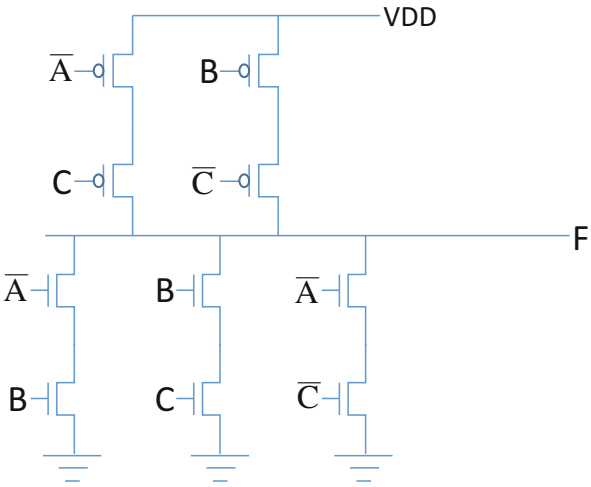


Fig. 6.19 CMOS circuit for exercise 6.20

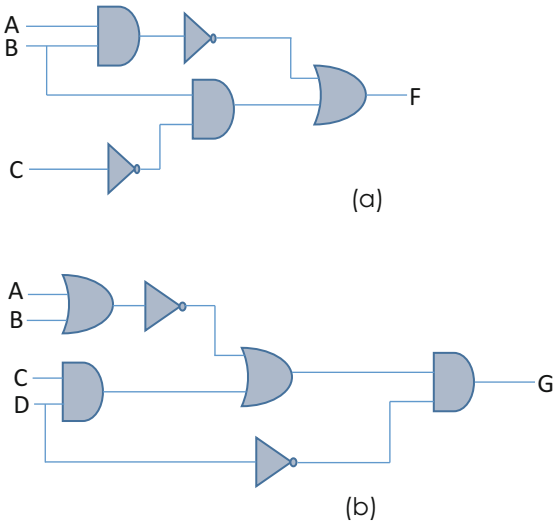


Fig. 6.20 Logic gate diagrams for exercise 6.21

6.19 Implement each of the following functions on an OR-AND PLA:

(a) $F = AB\bar{C} + A\bar{B}\bar{C} + \bar{A}BC + \bar{A}\bar{B}C + ABC$
(b) $F = (A + \bar{B} + C)(\bar{A} + B + C)(A + \bar{B} + \bar{C})$

6.20 What logic function is implemented by the CMOS circuit in Fig. 6.19?

6.21 Give the logic function corresponding to each of the gate diagrams in Fig. 6.20.

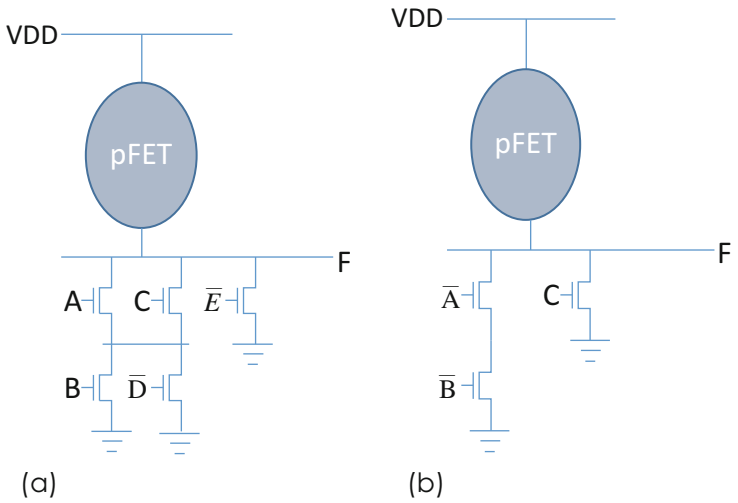


Fig. 6.21 CMOS transistor arrays for exercise 6.22

6.22 Each of the diagrams given in Fig. 6.21 show only the nFET portion of the CMOS circuit. Figure out which function is being implemented and draw the pFET portion.

6.23 Draw the gate-level implementation diagram for each of the following logic functions.

(a) $F = A\bar{B}(C + D)$
(b) $F = A + B(C + \bar{D})$
(c) $F = X + \overline{(Y + Z)}$
(d) $F = \overline{(X + Y)} + \bar{Z}\left(X + \overline{WY}\right)$

6.24 Give the logic equation corresponding to each of the gate-level diagrams shown in Fig. 6.22.

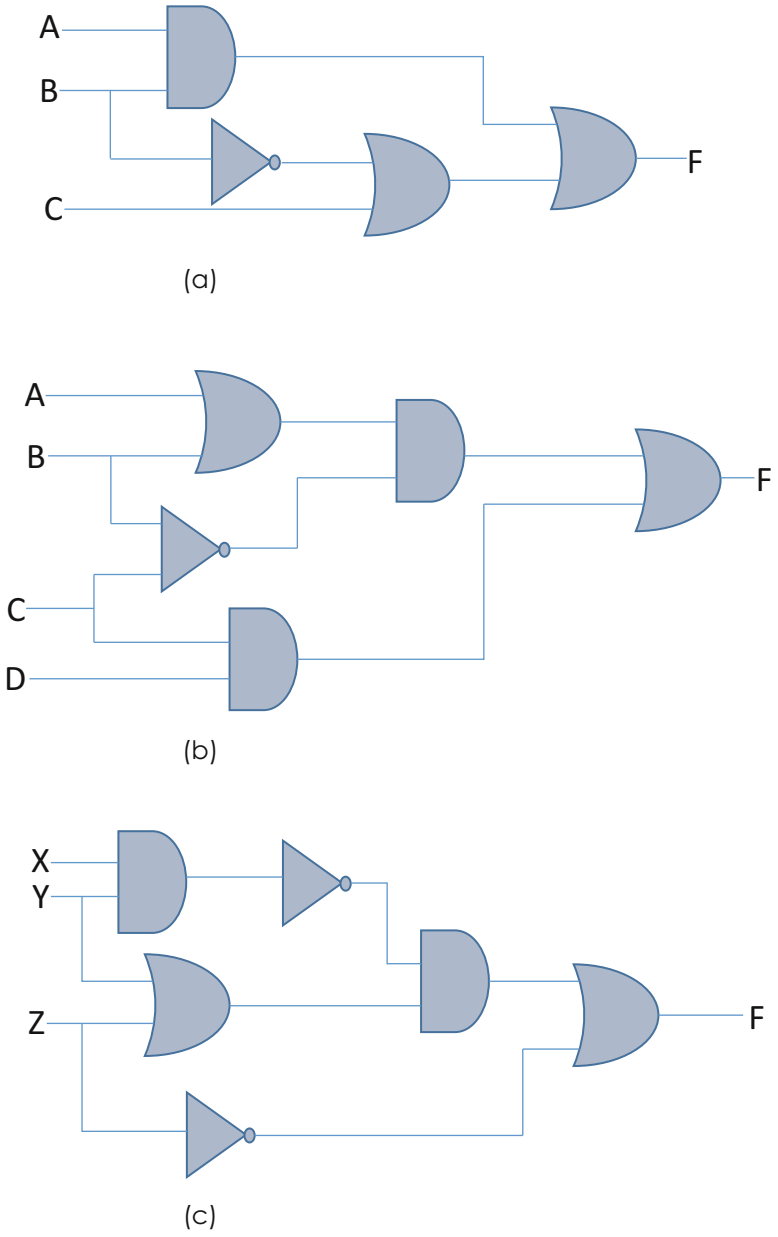


Fig. 6.22 Logic gate diagrams for exercise 6.24

Chapter 7
Unsigned Arithmetic

We've come pretty far. We've established that in our digital electronic computer we are going to use values that take on one of two values and found a mathematics called Boolean Algebra that supports this. We know how to specify logic functions within this mathematics to describe our system and we even know a bit about the synthesis process whereby the specification is converted into a physical layout. It's time to zoom out a bit from the transistor level and start using our core knowledge to actually get about the business of designing the necessary components to build our computer. Since we need our computer to be able to actually, well, *compute*, we're going to start with some arithmetic.

Unsigned Binary

If our computer is to actually add and multiply numbers, we need to figure out how to use these logic-1 and logic-0 values to represent numbers and to work with them. It turns out that exactly how we represent numbers is a design decision based on the available technology and is influenced by what we want to do with the numbers. In our everyday life, we use numbers to represent quantities, values such as times to be displayed on a digital clock, and even simply as labels. Each use for a number lends itself to a different way of using logic-1's and logic-0's to represent it. (This is actually a tremendously important point that you should make sure you get—too often we assume numbers in computers are "just 1's and 0's" when in reality the exact way we configure the 1's and 0's for a given purpose is a challenging design problem in and of itself.)

Often when we design arithmetic components in a computer, we first ask ourselves how we, as humans, would approach the task. If we do that here we remember the familiar addition algorithm we use when working with decimal numbers. We first add the 1's digits, then apply the carry to the next column, add those digits, and repeat. This process is shown in Fig. 7.1.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_7

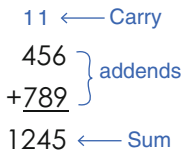


Fig. 7.1 Basic addition algorithm

We are able to do this because the numbers we are familiar with are in a **place-value system**. This means the position of each digit in the number affects its value in an organized way, namely, the appropriate power of the base.

$$456 = 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

Other systems we may be familiar with, such as roman numerals, are not place value systems. Consider the roman numerals XXIV and VIII. Each has an I in the position second from the right, but how that I is interpreted is different in each one. In XXIV we subtract the I from the V to its right, while in VIII we add the I to the VI to its left. Roman numerals are a great representation to use for *displaying* important numbers such as founding dates or *labeling* important positions such as monarchs, but it fails as a representation for *arithmetic*. For computations, we really ought to be working with place-value systems.

What place-value system can we implement with only two values? Since the **base** of any place-value system counts how many symbols we can use (base 3 would use only 0, 1, and 2 for example) we must use base 2. The special name for the place-value notation with base 2 is **binary.** (Note that in English the term *binary* usually means we have a choice between any two options and so it may seem that any representation made up of 1's and 0's is *binary* in that sense. We reserve the term *binary*, however, just for the specific representation involving place-value with two symbols.) More specifically, since we are working only with non-negative numbers, we call this representation **unsigned binary**. Signed numbers, those that can be positive or negative, are discussed in the next chapter.

Here is how it works: we interpret an unsigned binary number just like we would a decimal, except that instead of powers of 10 we work with powers of 2 (now would be a good time to memorize the powers of 2 up to about 2^{10}, by the way). For example,

$$11010 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 26$$

We can then add numbers just like we would in decimal, by starting with the one's bit (in binary, the digits are called *bits*). This process is shown in Fig. 7.2.

We distinguish the carry out from the leftmost bit from the other carries by calling it the **end carry.** It has special properties we'll discuss later.

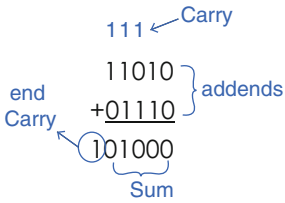


Fig. 7.2 Binary addition algorithm

435	= 110110001
216	1 --least significant bit
108	0
54	0
27	0
13	1
6	1
3	0
1	1
0	1
0	0—most significant bit

Fig. 7.3 Dividing-by-two algorithm for Dec to Bin conversion

How do we get the unsigned binary representation of a particular decimal number? We can just look at and keep adding powers of two together like a puzzle until to all clicks, but we're in computing and would rather have an actual algorithm to use.

In order to convert from any base to any other base, you keep dividing the number by the target base as represented in the current base and track the remainders. So, to convert from base 10 to base 2 you divide the base 10 number by 2 over and over again and write down all the remainders until you keep getting 0 every time. If one wanted to convert from base 10 to base 7 you would just divide successively by 7 and track those remainders. (Interestingly, one way to convert from base 2 to base 10 is to divide the base 2 number by 10, which is 1010 in base 2, and tracking those remainders. That's typically harder to do by hand than just adding up the powers of two, though. But it's much more awesome.)

For example, if we have the number 435 in base 10 and want to find its binary equivalent we run the process illustrated in Fig. 7.3.

When you get to the point where you have 0 remainder 0 you are done. Any other divisions will keep being 0. The last number you get is the **most significant bit** of the new base 2 number while the first number you obtained is the **least significant bit** of that number (so, it doesn't matter how many leading zeroes you add.)

Now we are ready to design our adder!

Full Adder

Each of our logic gates gets a special symbol, as do circuit elements such as resistors and inductors in analog circuit diagrams. We're going to be making so many components that we're not going to use a special symbol for each and every one of them. Therefore, we'll see a lot of the block diagram in Fig. 7.4 as the generic design for our devices.

For each device, we need to decide on its **inputs, outputs,** and **controls**. The inputs are the main data we're working with, the outputs are the results of our operation, and the controls can choose various modes. If we wanted this to both add and subtract, for example, we would have a control signal to choose between the two options. Since we're just adding here we won't have a control in this design. But we do have inputs and outputs. What are they?

Well, we know that we at least need two bits for input and a sum bit for output. This is shown in a block diagram framework in Fig. 7.5.

This works for $0 + 0 = 1$, $0 + 1 = 1$, and $1 + 0 = 1$, but what about $1 + 1 = 10$? We need two bits for this. We actually need a carry bit in addition to the sum bit. Figure 7.6 shows the corrected version.

We're all set now to add two 1-bit numbers together. But, how exciting is that? Not very. We really want to be able to add large 32- and 64-bit numbers together. Drawing on our knowledge of abstraction and modularity from our computer science programming skills, we can see that when performing addition we're repeating a basic operation many times: for each position, we add the two bits as well as the potential carry from the previous position. So, we can envision our large 32-bit adder being made up of 32 one-bit adders all chained together (Fig. 7.7).

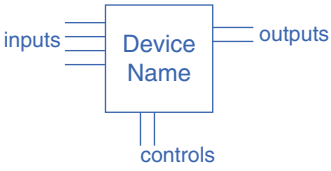


Fig. 7.4 Generic digital device block diagram

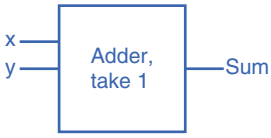


Fig. 7.5 Two-bit adder block diagram without carry out

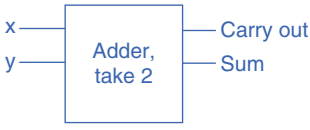


Fig. 7.6 Half adder block diagram

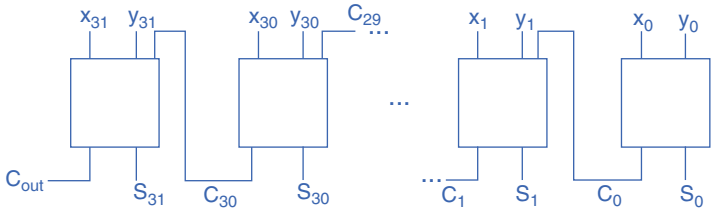


Fig. 7.7 32-bit ripple carry adder

Fig. 7.8 Full Adder block diagram

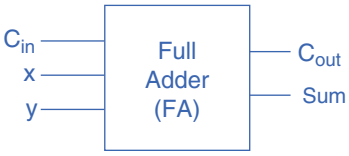


Table 7.1 Truth table for the Full Adder

C_{in}	x	y	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

We can see the 32-bit x and y numbers being added at the top and the 32-bit sum s arrayed at the bottom. We also see the internal carries feeding into the next adder and the final end carry c_{out} as an output from the entire system. Go over this until you are sure you can tell this story and really see the connection between what you do when adding decimal digits and the above diagram which demonstrates how that algorithm can be implemented in hardware. (Yes, that's actually key here: we are building a physical device that runs an algorithm. This is a theme of this text and we'll be doing a lot of this. If you are a CS-type interested in lots of algorithms, this is exciting.)

We see that we need one more tweak to our adder diagram. Instead of just adding two bits together, we really need to add three to account for the carry from the previous column, as seen in Fig. 7.8.

We call this a **Full Adder** (abbreviated **FA**) and use it as our basic building block in arithmetic circuits of virtually all types. If you remove the c_{in} input you get a Half Adder (HA) which isn't used very often (though it could be used in the rightmost bit of our 32-bit adder above, as we don't have a carry in to the entire system.)

Let's build the truth table for the full adder and specify the *sum* and c_{out} functions. Table 7.1 shows this beauty.

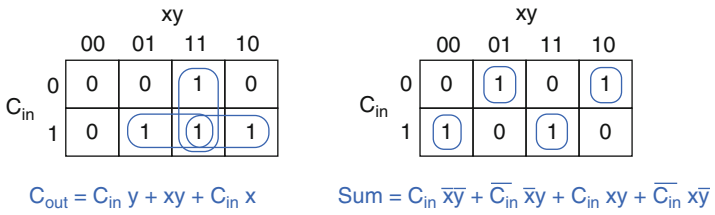


Fig. 7.9 K-maps for the Full Adder

You can verify these values by adding the three inputs together and seeing the corresponding outputs.

We can now synthesize the functions, perhaps using K-maps to get minimal forms (Fig. 7.9).

If we inspect these functions, we can see that the form of c_{out} makes intuitive sense: we have a carry out if any two of the inputs are 1. The *sum* equation is disappointing: the minimal form is the same as the canonical form! The K-map didn't help at all! Alas, such is the life of a designer. What can you do?

The final step for us is to derive the gate-level implementations and then continue and inspect the transistor-level implementation. If we were more focused on the electrical engineering elements of digital design we could optimize this circuit a great deal. As our goal here is to use components to construct computers rather than optimize the transistor layouts of the components themselves, we'll consider it a job well done once we get the gate diagram. But, for those interested, more electrical-engineering focused digital design resources abound.

It's worth noting that there is also such a thing as a Half Adder (HA) which only adds two bits together and generates a sum and carry out. While these can be used in specialized applications, we usually are in need of adding three bits due to the nature of place-value representations and the necessity of accommodating the carry bit.

High-Speed Adders

The adder we have built is called a ***ripple-carry*** adder as the carries *ripple* through the FA elements. The time it takes to perform an addition is a function of the number of bits in the numbers, as you can't compute the 5th position's addition before you know whether the 4th position has a carry. This follows our intuition from the basic algorithm we typically learn in grade school for how to add numbers.

Sometimes our intuition helps us when designing circuits, but sometimes it can limit us. In this case, for example, we need to think bigger and faster.

The computer can do something we cannot—it can attend to multiple computations at one time. If we knew ahead of time exactly where the carries were going to be, then the circuit could compute **all 32 bits at once!** The time it takes to perform

arithmetic would no longer depend on the number of bits in the number! We would end up with a potentially much faster adder.

But from what crystal ball do we divine the future: how can we possibly know where the carries are going to be ahead of time? To answer that, let's consult the oracle of mathematics and very carefully analyze exactly what it is we are doing.

We can reason that if both bits in a given position are 1, then a carry will be **generated** at that position. We can also reason that if at least one bit in a given position is a 1 then a carry from the previous position will be **propagated**. Formalizing these two ways to transmit a carry, we can see that.

$$\textbf{Generate} \quad g_i = x_i y_i$$
$$\textbf{Propagate} \quad p_i = x_i + y_i$$

The carry bit, which is a 1 if a carry was generated in the previous position or if a carry was propagated from the position two previous, can be captured in the equation.

$$\textbf{Carry} \quad c_{i+1} = g_i + p_i c_i$$

With this, we can do some algebra and compute what the carry bits are going to be:

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0)$$
$$c_3 = g_2 + p_2 c_2 = g_2 + p_2(g_1 + p_1(g_0 + p_0 c_0))$$
$$c_4 = g_3 + p_3 c_3 = g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0 c_0)))$$

Every carry bit c_i can be computed only from generates and propagates as well as the original carry in c_0. Since the generates and propagates only depend on the bits of the addends, and we know c_0 at the outset of the calculation, we can compute all the carries in one fell swoop at the start of the addition! Then, once we know the carries for each position, we can use the inherent parallelism of digital circuits to add **all the bit positions at one time!** It won't matter if we have 4 or 16 or 32 or 64 bit numbers. They will be added at the same speed once we know the carries! This is called a **carry-lookahead adder**.

Why don't we do this by hand? Well, as the algebra for the c_i's makes clear, it takes more calculations for us to find all the carries than it would to just get on with the business of adding. But we can optimize the computation of the carries inside the circuit and greatly speed up the performance of the adder. Yes, the carry lookahead process does depend on the number of bits in the number, but it's a much faster process, when properly built, than the ripple-carry approach. It turns out that for number larger than about 6 or 7 bits the carry-lookahead overhead is worth it. That is, the amount of time taken to compute the carries is offset by the speedup gained by performing the actual additions in parallel.

To build high-speed arithmetic components of all types we really need to study what is going on mathematically. This is an important example in that it shows the

power of mathematical modeling to achieve performance gains in our digital systems. Much of the field of **computer arithmetic** is like this. It gives a designer a chance to work out a great and powerful algorithm and then see its realization in a physical device. Hardware design is not all about the soldering and board layout—there is plenty of pure computer science embedded within.

Subtraction

How do we subtract unsigned binary numbers?

How do we perform this subtraction? Well, we borrow of course! Remember borrowing? That happens when you have something like Fig. 7.10.

You borrow a 10 from the 5 (which is really 50) and add it to the 6 to get 16. Now you can subtract the 9 from the 16 and proceed with the computation. We do the same thing in binary in Fig. 7.11.

We borrow a 2 from the 1 (which is really 2) and add it to the 0 to get 10 (which is 2 in binary) so we can subtract 1 from the 10 and get 1. Wow. That's a mouthful. Make sure to go through this so you understand.

As seen in Figs. 7.12 and 7.13 (decimal and binary), sometimes we have to borrow across multiple positions.

What happens in binary is that when you borrow you always end up with a 10 because you start with a 0 (in base 2 there are fewer options to consider! It's actually easier in many ways once you get used to it.) When you have to borrow across multiple positions, then, you always end up with a 1 in the positions that were originally 1's. This operation will be especially relevant in the next chapter when we look at computing the two's complement of numbers.

Why don't we build a subtractor? We use signed numbers. . .. . .next chapter!

Fig. 7.10 Basic subtraction algorithm

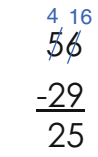
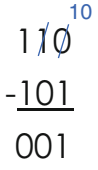


Fig. 7.11 Binary subtraction algorithm



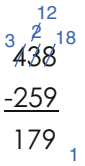


Fig. 7.12 Borrowing across multiple positions

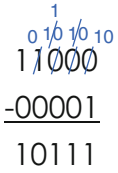


Fig. 7.13 Binary subtraction with multi-bit borrowing

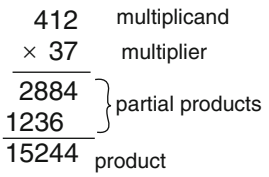


Fig. 7.14 Basic multiplication algorithm

Multiplication

How do we multiply unsigned numbers? Let's first recall how we multiply decimal numbers. Consider 412×37 as seen in Fig. 7.14.

We see that multiplication can be broken down into two steps: finding partial products and then summing them. It's important here to familiarize yourself with and to be able to use the technical language surrounding this computation: the **multiplicand** is the "top" number, the **multiplier** the "bottom" number, the **partial products** are the result of scanning the multiplier digit by digit and performing multiplication, and the **product** is the sum of the partial products.

It's worth considering the difference between the multiplicand and the multiplier. We know multiplication is commutative, that is, that $x \times y = y \times x$, so why should it matter which we call the multiplicand and which we call the multiplier? The answer lies in the *algorithm*. Yes, it's true that we can multiply numbers in any order and arrive at the same product either way, but the *procedure we use to perform the multiplication* does in fact depend on the ordering. In the example above, if we multiplied 37 by 412 we'd have three partial products to sum instead of two. We use the digits in the multiplier differently than we use the digits in the multiplicand. This level of precision in thought and language helps us build fast computer arithmetic circuits. It will be particularly relevant when we work with signed arithmetic in later chapters.

We generate the partial products by multiplying each digit of the multiplier by the entire multiplicand, and shifting each result to correspond to the place-value position of each digit.

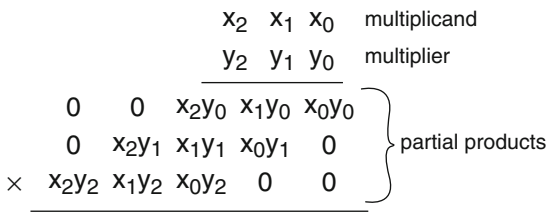


Fig. 7.15 3-bit array multiplication

The thing that makes it hard to do when we learn is that the multiplications themselves are tedious. In our example, we have to know our 7's tables pretty well as we form the first partial product. In a larger problem, we'd have to bring to bear all our base-10 multiplication knowledge just to get the partial products. This is one of the areas were unsigned binary is actually *easier* to work with than the decimal. All we need to know are the 1's and 0's tables! We never have to multiply by something as exotic as 7! If the multiplier bit is 1, we simply copy the multiplicand as it is, and if the multiplier bit is 0 we have no partial product to worry about at all! Starting with our knowledge of how we multiply in decimal, we've constructed an algorithm and then seen the ease with which we can rotely carry out the operation in binary. This happens a lot in computing.

Figure 7.15 shows what a 3-bit multiplication looks like when x and y are unsigned binary numbers:

We use 0's to pad both the beginning and end positions of the numbers so that they all have the same number of bits (which is required for our adder.) The first partial product is just the individual bits of x multiplied by y_0, the second partial product is the individual bits of x multiplied by y_1, and the third partial product is the individual bits of x multiplied by y_2. We don't have to know any times tables or perform any carrying in unsigned binary like we do in base 10 because all we ever do is multiply by 1 or 0 and neither of those numbers require such complications.

How do we perform the bitwise multiplications? If we look over our logic functions, we see that the AND gate performs exactly the same function as we expect our arithmetic multiplication to perform, so we can just use it directly. This is a nice consequence based on how we set up our truth table for AND. It doesn't obtain for OR, as we needed to go through a lot of work to build our adders. But bitwise multiplication is exactly the logical AND operation, so all we need are AND gates.

Let's build the multiplier circuit! We have three numbers to add together, so we'll need two rows of adders to accomplish this. If we shift correctly and put the 0's where they are supposed to go, we have the circuit seen in Fig. 7.16.

This is called an **array multiplier** because of the way it's organized. It is based on the ripple-carry adder and is therefore of O(n) time complexity. We can speed this circuit up in many ways. The computer arithmetic chapter discusses more of them in depth. For now, this is good enough. Make sure you review this discussion and see how we convert a basic algorithm we're very familiar with into a logic circuit design. This is about more than just some arithmetic—this speaks to the

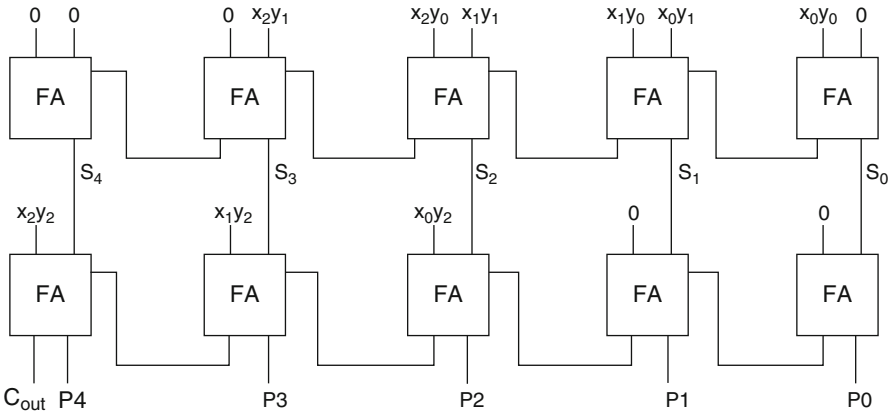
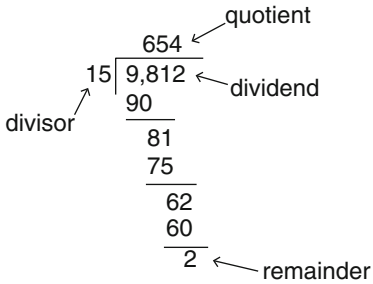


Fig. 7.16 Array multiplier

Fig. 7.17 Basic division algorithm



heart of the digital logic design process itself. It's all about taking an algorithm, in any form, and implementing it with our digital components instead of with the `for` loops or `if` statements we're used to from our high-level programming languages.

Division

Long division is everyone's favorite mathematical operation! Books are sold with long division diversions, poets write about it, and many are the songs written in its celebration. Figure 7.17 presents the core of it.

Just as with multiplication, division has its own terminology: we divide the **dividend** by the **divisor** to get the **quotient** and a **remainder**. Unlike multiplication, division is not commutative, so it may be more clear from the beginning why having precise names for all the numbers involved is desirable.

The thought process behind division is familiar: we ask how many times does 15 go into 98 and write down our answer, 6. We multiply 6 by 15, subtract from 98 and get the difference, 8. We bring down the next digit, 1, and repeat: how many

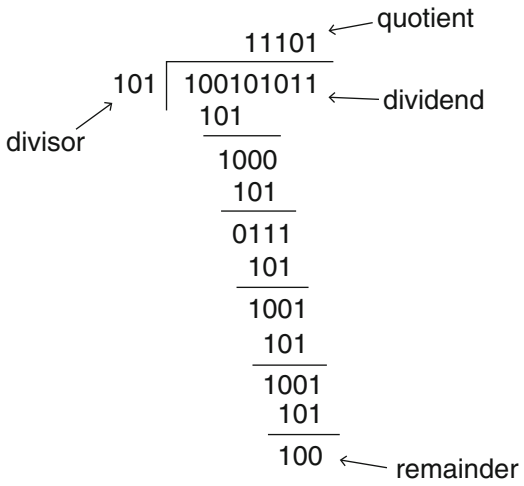


Fig. 7.18 Binary long division

times does 15 go into 81? We answer "why, five times, of course," multiply 15 by 5, subtract, bring down the next digit, and so on until we have a number smaller than 15 which we call the remainder.

Any difficulty we may have in implementing this algorithm lies in the core question of its basic operation: how many times does x go into y? We have to approximate, guess, double check, and so on until we've solved that puzzle. Then we perform some rote opeations and we're back to the question again. In Fig. 7.18, we see this process in action for unsigned binary numbers.

Now our repeating question begins "how many times does 101 go into 1001?". This may seem harder because we're not yet used to binary and have to translate in our heads and we might be thinking "this means how many times does 5 go into 9", but if we stop and reflect for a moment we come to a great epiphany: *how many choices do we have*? In base-10 when we ask how many times 15 goes into 98 we have 10 potential answers. But in unsigned binary when we ask how many times 101 goes into 1001 we have only two potential answers: 1 or 0, either it does or it doesn't. All we have to do is check to see whether the divisor is less than or equal to the number we're comparing it to. If it is, then we write down 1 and if it not then we write down 0. It's incredibly easy. So easy, in fact, that a computer can do it.

And this is the crux of it all: we've converted our complex decimal long division problem into a rote mechanical binary comparison problem. This lets us harness the power of our digital logic circuitry to perform this operation. Again, as with multiplication, the takeaway here is far more than just "this is one way to implement division in a circuit". It's about the wider idea of translation of what we're doing in our human mind into what the electronics are capable of doing. We usually take our own familiar approach as the starting point and then we tweak with cleverness and mathematical sophistication until we arrive at an algorithm only achievable through the inherent speed and parallelization of digital logic.

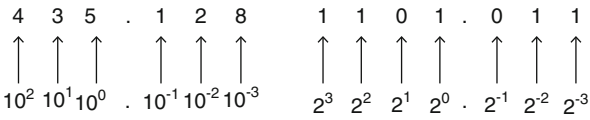


Fig. 7.19 A demonstration of place value

We'll explore this idea more when we discuss restoring and non-restoring division algorithms in our computer arithmetic chapter. For now, familiarize yourself with the basic process of dividing binary numbers, and you'll be well prepared when we get to the actual circuit implementation.

Fractions

If we were to continue the division process, we'd soon need to represent fractions in unsigned binary. How do we do that? If we compare again unsigned binary against decimal numbers, we can extend the place-value concept to the right of the one's position, as seen in Fig. 7.19.

What we call the *decimal point* in base 10 is called the *binary point* in base 2 and is called, more generally, the *radix point*. Digits to the right of the radix point are still in place value. They begin with the power of -1, then proceed with -2, -3, and so on.

The binary number 1101.011, for example, is interpreted as 8 + 4 + 1 + .25 + .125 = 13.375. We can also see that, to the right of the radix point we have 011 and the least significant bit represents $2^{-3} = 1/8$ so what we're working with is 3/8ths, or .375. And this is why you memorized your fraction-to-decimal eights conversions! So it's clear that .101 = 5/8ths = .625!

Hexadecimal

Before we move on to how to work with negative numbers, it's worth spending a moment talking about a base other than 2 which is of great importance in computing.

We often need to work with binary numbers that are very long. Memory addresses, for example, can be 30 or 40 bits long. It's not practical for us to read or write bit strings of that length. Therefore, we've come up with a shorthand notation for representing such numbers. It's called **hexadecimal.**

Hexadecimal numbers are technically base-16 numbers. Since we only have symbols for the ten decimal digits 0–9, computer scientists needed to come up with six more symbols to represent the rest of the hex digits. Did they follow the path blazed by their physicist colleagues and conjure up evocative names such as charm and beauty for these digits? Sadly, no. They looked around their immediate vicinity

Fig. 7.20 Hexadecimal numbers

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

for some other symbol to use for inspiration. Apparently they didn't look far because the symbols chosen were just the ones right next to the numbers on the keyboard: simple letters a through f. Since there are 16 hex digits, that means each one corresponds to a 4-bit binary number. Figure 7.20 shows the hex system.

We encode large binary numbers by simply taking groups of four bits and replacing them with their hex equivalents. For example:

$$1100101010000111 = \text{xCA87}$$
$$100001010110001 = \text{x42B1}$$

Notice in the second example we had to add a leading 0 to make sure the total number of bits was a multiple of 4. The notation for hex usually leads with an x before the number. It's a C thing.

Be assured this is not a special exception to our normal rules for converting from one base to another. This is actually an application of our standard iterative division method that we've used to convert from base 10 to base 2. We divide our base 2 number by the target base, 16, represented in base 2. Since $16 = 1000$, dividing a base 2 number by 16 is really easy. You just shift the radix point to the left four places. We then record remainders and write down the number. That's exactly what we did. We're just good enough to do the division in our heads.

Converting back from hex to binary is just as easy, simply unpack the hex encodings:

$$\text{x5E38} = 0101111000111000$$

This is an excellent time to stress the hex encodings should be memorized. It will save you tons of time and effort and make you a more fluent computing professional when it comes to working with, reading, and writing hexadecimal numbers.

Being actual numbers, we can also perform arithmetic on hex numbers.

Exercises

7.1 Perform the following binary additions:

```
 11010101   01011010   10101111   01011010
+01010111  +11100010  +10001110  +11100101
```

7.2 Perform the following binary subtractions:

```
 11001010   11000101   10000000   10001010
-10110110  -00001101  -00000110  -00010101
```

7.3 Perform the following hex additions:

```
  A2    B21E    ABCD    2539
+2F   +1CD2   +FEDC   +8922
```

7.4 Perform the following hex subtractions:

```
  A9    2BE    4AC2    492
-15   -1AF   -2D88   -236
```

7.5 Perform the following octal additions:

```
  34   177   327   4213
+56  +275  +122  +5371
```

7.6 Perform the following octal subtractions:

```
  52   241   561   45210
-17  -132  -321  -35024
```

7.7 Perform the following base six additions:

```
  35    421
+154  +478
```

7.8 Perform the following base nine additions:

```
458   182
+24  +355
```

7.9 For each of the following additions, indicate what base is being used:

```
325   261   359
+42  +543  +8B6
411  1134  1053
```

7.10 Convert the following decimal numbers to binary: (a) 214.75, (b) 1586.125, (c) 78.40.
7.11 Convert the following binary numbers to decimal: (a) 10110.010, (b) 11001101.11, (c) 110011010010.1101
7.12 Convert the following decimal numbers to hex: (a) 62.875, (b) 112.25
7.13 Convert the following hex numbers to decimal: (a) 4AB2.C, (b) 12FFE.B1
7.14 Convert the following hex numbers to binary and octal: (a) 3BCDA.2, (b) F0F1.EA
7.15 Convert the following binary numbers to hex and octal: (a) 1011011011.0010101, (b) 101010.110100101
7.16 Convert the following decimal numbers to octal: (a) 78.375, (b) 180.875
7.17 Convert the following octal numbers to hex and binary: (a) 73210.23, (b) 10242.5
7.18 Convert the following decimal numbers to base five and seven: (a) 125.60, (b) 59.75

7.19 Perform the following unsigned binary multiplications:

```
1101  1011  1001  1010
0110  1101  0111  1010
```

7.20 Perform the following unsigned binary divisions:

a) Dividend = 100101110 b) Dividend = 100111010 c) Dividend = 1100010110
Divisor = 101 Divisor = 1101 Divisor = 110

7.21 Convert the decimal number 2043.72 to binary. Round to 4 significant figures.
7.22 Convert the octal number 1247702.216 to hexadecimal.
7.23 Convert the following from decimal to binary:

(a) 125
(b) 23981
(c) 193.128

7.24 Convert the following from decimal to octal:

(a) 3402
(b) 124
(c) 12.5

7.25 Convert the following from decimal to hexadecimal:

(a) 92
(b) 209
(c) 8762.23

7.26 Convert the following from binary to decimal:

(a) 10010
(b) 110.1011

7.27 Convert the following from octal to decimal:

(a) 3021
(b) 712.45

7.28 Convert the following from hexadecimal to decimal:

(a) 23FA9
(b) 912.B01

7.29 Convert the following from binary into both octal and hexadecimal.

(a) 110110
(b) 1110011100011

7.30 Convert the following from octal into both binary and hexadecimal.

(a) 2306
(b) 102.34

7.31 Convert the following from hexadecimal into both octal and binary.

(a) 24A90
(b) 10.D2

7.32 Design a device that will output a 1 if the input is a perfect square and a 0 otherwise. Assume the device takes a 5-bit input in unsigned binary format. Your solution should include a K-map and gate-level logic design diagram. You may find it helpful to work out a truth table for this problem, but it is not necessary if you can write the K-map directly.

7.33 A flying saucer crashes in a Missouri farmer's field. The Kirksville police investigate the wreckage and find a blue plastic disk containing an equation in the alien number system: $325 + 42 = 411$. What base does the alien number system use?

7.34 Use FA's and logic gates to build a device which will multiply two three-bit unsigned inputs.

Chapter 8
Signed Numbers

We've designed adders that can work with positive numbers. But what about negative numbers? We call systems that can incorporate negative numbers **signed representations** and consider those that cannot handle negative numbers to be **unsigned**. The binary system discussed in the previous chapter is more properly called *unsigned binary*.

To work with negative numbers, we need to figure out how to handle the sign. When we write *−18* to represent a negative number, we're actually cheating a bit because the negative sign – is acting sort of as an eleventh symbol. We're not really representing −18 in decimal! We're using the sign as a visual aid to tell us to subtract it from something instead of adding, but we're not indicating that the value is less than zero by the use of our digits 0–9. Inside the computer, we don't have the luxury of introducing a new symbol to act as a visual aid. All we have are 1's and 0's and we must use these two values, and only these two values, to represent negative numbers.

Sign-Magnitude

One approach is to use the leftmost bit position to indicate the sign of the number. To maintain consistency with our unsigned binary from the previous chapter, we'll make a 0 bit represent 7positive numbers (or zero). And we'll say that a 1 in the leftmost bit means the number is negative. Beyond this sign bit, we will interpret the numbers as we did unsigned binary. For example, we have in this notation the quantities in Fig. 8.1.

Breaking the number up like this into **sign** and **magnitude** bits is called **sign-magnitude representation.** It's a departure from our purely place-value representation in that now we have a bit, the leftmost bit, that is **not a place-value position**. That bit should not be part of computations based on place-value notation. This

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_8

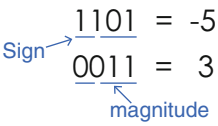


Fig. 8.1 Sign-magnitude numbers

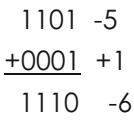


Fig. 8.2 Sign magnitude addition gone wrong

means we cannot just send the entire sign-magnitude numbers into a place-value based adder and expect the result to be accurate, as seen in Fig. 8.2.

We get that $-5 + 1$ is -6! That's because while we can add the magnitudes just fine, the sign bits must be handled in a special way. This is reminiscent of what we see as humans when working addition problems. When we see $-5 + 1$ in decimal, we don't even consider *adding* the five and the 1 together. Instead, we take the negative sign on the -5 as a cue that we ought to *subtract* the magnitude of the negative number, in this case 5, from the magnitude of 1, the positive number. In another case, if we have $-5 - 1$, then we *do* add the magnitudes of both negative numbers together, just as we would in the case of two positives like $5 + 1$. So, what we actually have here is a decision process that must take place prior to any adding. We must decide both *whether to add or subtract* and *if we're subtracting, what do we subtract from what*.

In practice, what this means for our digital system is that an adder designed to process sign-magnitude numbers must have a preprocessor which first looks at the sign bits and then a normal adder/subtractor which will perform the required calculation. This is a much slower process than what we witnessed in the previous chapter. The preprocessing takes time and in the current technological environment that is time we do not want to be spending. We want something faster. Thus we are motivated to design a different representation for negative numbers.

This story is important—make sure you can tell it to your non-computing friends at the lunch table. It's key to understand that our signed number representations—really, all the ways we encode numbers within the computer—are results of complex well-thought-out design processes with specific goals in mind. There was a time when sign-magnitude numbers were used. The reason they fell out of favor was that as the underlying technology shifted to permit faster and faster circuitry, the overhead of the required preprocessing was no longer worth the benefits of the intuitive nature of the representation.

So, sign-magnitude won't work. What will?

Two's Complement

It's the preprocessing, really. We need to cut that. We need a system that doesn't require such overhead. We want to simply run whatever numbers we have, be they signed or unsigned, through the same arithmetic hardware and rest secure that the results will be correct.

The key to this is to build a signed representation that is entirely place-value, because our unsigned representation is entirely place-value and our adders, detailed in the previous chapter, are constructed with that feature in mind. It would take an entirely different algorithm to add numbers that were not in place-value notation. We don't want to go back to the drawing board. We want to use the adders we have.

So, what do we do? We don't want to throw out all the ideas from sign-magnitude. We liked that there was a sign bit so that we could quickly determine whether the number was negative or not. Let's keep that, and say that all number that start with a 1 are negative. That's good. But we need this sign bit to represent part of the value of the number. It can't just be a visual cue as the negative sign is when we write our decimal numbers like -24. It must be part of the magnitude. What is the solution? Think about it for a minute before proceeding, because the solution is sublime and aesthetic in character.

Remember that the leftmost bit in a number, the one we are reserving for the sign, is called the *most significant bit*. This is because the bit in unsigned binary carries the largest value—it is assigned the largest power of 2. If we think what that means, we can see that all the remaining bits, taken together, can never sum to the same value the most significant bit signifies. For example, if we have a four-bit unsigned number the most significant bit represents $2^3 = 8$ while a 111 in the three remaining bits is only equal to 7. This gives us our in—we can make the most significant bit, the *sign bit*, simply equal to the negative of 2 to whatever power we're using. In the number 1110 we can let the most significant bit represent $-2^3 = -8$ and rest assured that the remaining bits will never sum to a number so large that -8 plus that number is no longer negative.

That's it! That's the solution! Make sure you follow, because it's so simple in retrospect that it's easy to fail to fully appreciate the artistry and cleverness involved in designing it. This system, called **two's complement**, is universally used today and yet it took decades of computer science development before the computer engineers realized its significance.

To recap, in the signed notation called two's complement, a number is interpreted almost exactly as if it were unsigned—the only difference being that we tack on a negative sign to the most significant bit. So, we have.

$$11001010 = -2^7 + 2^6 + 2^3 + 2^1 = -54 \text{ and } 110110 = -2^5 + 2^4 + 2^2 + 2^1 = -11$$

Table 8.1 illustrates the full four-bit representation:

We can see important patterns in this table. First, all three representations coincide for the positive numbers. When the sign bit is 0, the magnitude takes over and that's all there is. Both representations have a sign bit and all the other bits

Table 8.1 4-bit representations for unsigned binary, sign-magnitude, and two's complement

x_3	x_2	x_1	x_0	Unsigned binary	Sign-magnitude	Two's complement
0	0	0	0	0	0	0
0	0	0	1	1	1	1
0	0	1	0	2	2	2
0	0	1	1	3	3	3
0	1	0	0	4	4	4
0	1	0	1	5	5	5
0	1	1	0	6	6	6
0	1	1	1	7	7	7
1	0	0	0	8	0	−8
1	0	0	1	9	−1	−7
1	0	1	0	10	−2	−6
1	0	1	1	11	−3	−5
1	1	0	0	12	−4	−4
1	1	0	1	13	−5	−3
1	1	1	0	14	−6	−2
1	1	1	1	15	−7	−1

Fig. 8.3 Two's complement addition

```
 0101
+1100
10001
```

are magnitude bits. Where sign-magnitude and two's complement differ is in what value the magnitude is added to. Sign-magnitude adds the magnitude to 0, and two's complement adds the magnitude to -2^{n-1}. You can see that in the table where in sign-magnitude the low negative numbers are associated with the numbers with low magnitudes, whereas for the two's complement representation the low numbers are associated with high magnitudes. (Also, we have two 0's in sign magnitude—that's awkward!)

We're effectively taking what, in unsigned binary, are the largest positive numbers and making them the negative numbers in a particularly clever way. By choosing to represent the largest positive numbers by the smallest negative binary strings we take advantage of the fact that the numbers will roll over. Consider Fig. 8.3.

In unsigned binary, we're saying 5 + 12 = 17. The end carry bit that is generated indicates **overflow** has occurred in that system. Overflow happens when the result of an operation falls outside the expressible range of the representation. Our four-bit unsigned binary system can only express numbers 0–15, so 5 + 12 generates overflow. It's easy to detect: just look for the end carry bit.

If these are two's complement numbers, however, we now are looking at 5–4 instead of 5 + 12. We want 5–4 to be 1, and if we look at the four-bit sum, ignoring the end carry, we see that indeed the sum is 1. If you look at the table, you can see that when we add past the end of the table, we start over again. So adding 1100 to a number will count 12 lines from where we start to the end of the table and then go back to the top of the table and start counting again. This process ends 4 lines higher than what our original number is. In this way, by assigning the smallest negative two's complement numbers to the largest positive unsigned numbers, we are able to achieve subtraction within the exactly same adder that is mindlessly adding.

This is key: the hardware doesn't know what representation we are using. We want to use both unsigned binary and two's complement on the same arithmetic units. To do this, we only have to track overflow separately. We know if we get an end carry bit then unsigned binary will overflow. But that doesn't indicate two's complement overflow. What does?

Notice that adding any positive and negative two's complement numbers together results in a number within the expressible range of the representation. $-8 + 1 = -7$ and $-1 + 7 = -6$. It's always OK to add number with opposite signs together. We run into problems when we try to add two positive numbers ($5 + 4 = 9$ which is too big) or add two negative numbers ($-6–4 = -10$) which is too small. This means we will have overflow in two's complement arithmetic when we are adding two numbers of the same sign and get a sum (not counting the carry bit—the sum is the same number of bits as the addends) that is of the opposite sign.

Examples can be seen in Fig. 8.4.

Detecting overflow is critically important within our computer, as we need to know when a computation malfunctions. We track both end carries and two's complement overflow in special storage locations inside our computer so that our code can access these values and react if necessary.

Why is it called *two's complement*? Great question. Let's look at how we calculate the *complement*, that is, in informal language, *the negative*, of a number.

If we have the two's complement number 11001, how do we complement it? (And yes, this is confusing terminology! If a number is in *two's complement representation* or if it's a *two's complement number* or even a *signed number* this means the number is following the rules of the two's complement system with a weighted negative sign bit. If we want to find the *two's complement of a number* or just the *complement of a number* then we want to find the negative of that number

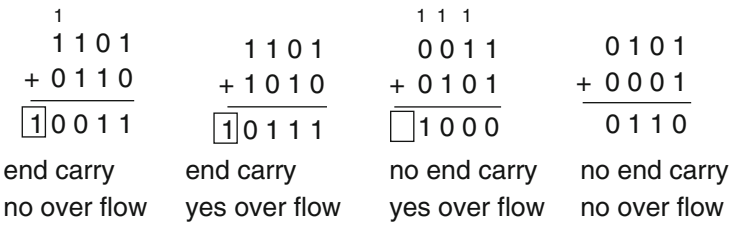


Fig. 8.4 Two's complement addition examples

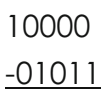


Fig. 8.5 Two's complement arithmetic set up

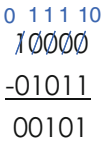


Fig. 8.6 Two's complement subtraction with borrowing

within the two's complement system. So, saying "find the two's complement representation of −7" results in an answer of 1001 while saying "find the two's complement of −7" results in an answer of 0111. Yeah, welcome to computing!)

We can think of an n-bit two's complement number itself as $-2^{n+1} + magnitude$. So, 1011 for example is $-8 + 3 = -5$. We want to figure out an algorithm that gives us the complement of this, or 5. We see that 8 + 3 = 11 which is 5 less than 16, which is the next higher power of 2. So, to get the complement of a two's complement number we start with the next higher power of 2, which is 2^n, and then subtract the sum of the number *as if it were unsigned*. We toss out the fact that we treat the most significant bit as the sign bit. We just subtract as if it were an unsigned number. To signify this we add a 0 to the left to make the number $n + 1$ bits. We end up with Fig. 8.5.

Using the subtraction method from the previous chapter, we end up with Fig. 8.6.

This gives us 5 just as we had hoped. Let's think about how to optimize this process. The fact that we are always subtracting from a 1 following by $n - 1$ 0's means we will always be borrowing all the way over to the first 1 in the number we're complementing. The fact that all the intervening positions become 1's while the position receiving the borrow becomes 10 also helps. Since this is binary, we know that subtracting a bit from 1 gives us the complement of the specific bit: 1–1 = 0 and 1–0 = 1. So we can just go ahead and, starting with the first 1 in the number to be complemented, "flip all the bits", or "complement all the bits", and then we are done.

For example, to find the two's complement of the signed number 10110101100 we don't have to do a bit of actual math. We just have to start with the rightmost 1 and complement each bit that comes after to arrive at 01001010100. Done. Easy. Great algorithm we have found.

There are other ways to look at it, too. You can also see that the 10 we get in the rightmost 1's bit position is actually 1 + 1 so that we can actually separate the 1 and the +1. Keeping the 1 in the number we're subtracting from lets us flip, or complement, *all the bits*, and then we just have to add the lingering 1 back in. So, the previous computation could also be rendered thusly: 10110101100 becomes 01001010011 + 1 = 01001010100 which is the same result as before.

```
010111  23          010111  23
-001111  15         +111000  -15
```

Fig. 8.7 Two's complement subtraction by addition of the complement

It's good to keep both these methods in mind, as there are times when designing digital systems that one will be more or less useful than the other. Taking the complement of numbers is important because we tend to avoid subtraction per se. In order to subtract we add the complement instead.

For example, 23–15 would be handled as 23 + −15 as shown in Fig. 8.7.

Building an Adder/Subtractor

Since we're going to go ahead and subtract by first calculating the two's complement and then adding the numbers, let's go ahead and augment our adder from before with the ability to do just that.

How do we calculate the complement of a number? We know we can complement each individual bit and then add 1. Adding the 1 is easy to do because if we use a Full Adder for the least significant bit position we have a carry in to the entire system that up until now we've had no use for. If we commandeer this bit to support the potential of adding the 1 in the event we are subtracting, we are halfway there. Now we need to figure out how to complement the individual bits. Do you remember how to do this? Which logic gate is employed to perform a bitwise complement?

It's the XOR gate! We'll XOR each bit of the subtrahend (in subtraction, we subtract the *subtrahend* from the *minuend*—great words to know and love) with 1 to help covert to the negative we need. We'll use the 1 that we're inputting through the carry in bit as the 1 we need to fuel the XOR gates to perform complementation. And, at long last, here is our adder/subtractor in Fig. 8.8.

We use *op* to label the carry in, where op =1 indicates subtraction and op =0 indicates addition. Eventually we can design more involved arithmetic units that can perform all sorts of operations. In this case, the op input will expand to cover however many functions we need. We can also think of it as an $\overline{add}/sub$ input, where the bar over the add indicates it's the operation indicated when the signal is 0 (active-low operation.)

Adder/Subtractors are often packaged in 4-bit units in integrated circuits (Fig. 8.9).

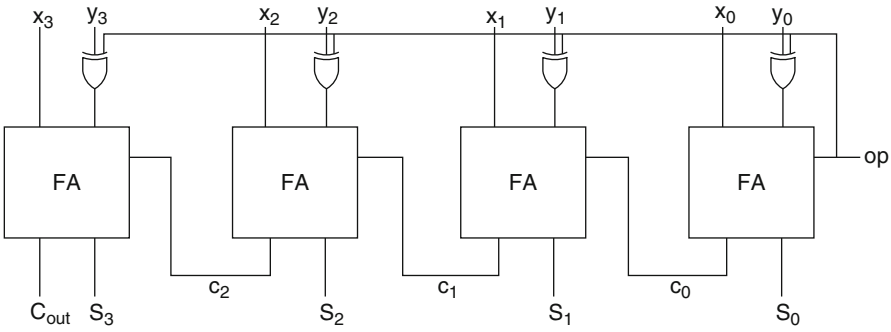


Fig. 8.8 Adder/Subtractor unit

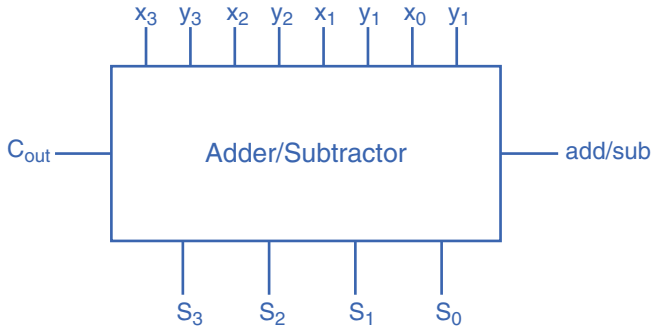


Fig. 8.9 Adder/Subtractor block diagram

Sign Extension

With these we can build adders that implement linear combinations of inputs rather than just $x \pm y$. Suppose, for example, that we want to compute $4x - 2y$, where x and y are 8-bit signed numbers. While we could use many arrayed adders to add x with itself 4 times, it's better to take advantage of properties of binary numbers and recall that we can multiply these numbers by shifting bits. To multiply by 4, we shift the number to the left 2 positions. To multiply by 2 we shift to the left one position. This is going to mean we need 10 bits for x and 9 bits for y. Our adder circuits are based on 4-bit units, so we'll need 3 of them for this computation. Our first pass at a solution looks like the diagram in Fig. 8.10.

Notice on the far right we've added 0's as inputs to the Add/Sub units to indicate the shifts necessary to compute 4x and 2y. We set the op input to 1 to indicate we are subtracting the y input from the x input. But look on the left-hand side. We have inputs left over in the third adder. What do we add to the *beginning* of these numbers to fill out the entire 12 bits?

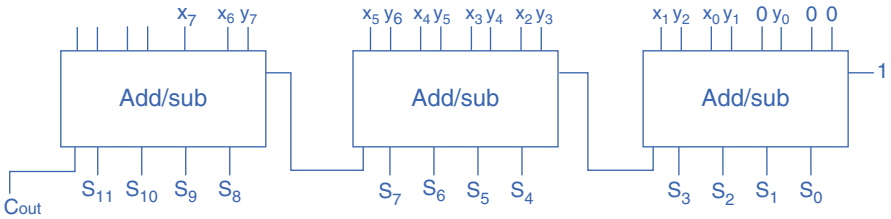


Fig. 8.10 Using adder/subtractor units to build larger ones

This is actually a common and important problem when working with signed numbers. Due to the way our hardware is constructed, we have to line up the bits when adding so that our numbers must be represented using the same number of bits. But often in application we find ourselves needing to add or subtract numbers represented with differing numbers of bits. We need a way to reconcile this, to *extend* a bit representation to a larger number of bits.

In unsigned binary, the solution is trivial: just add leading 0's. We know from the definition of place value notation that adding zeroes to the left of the most significant bit doesn't affect the value of the number. That is, the unsigned numbers 110, 0110, 00110, and 000110 are all the same value, namely 6.

But the same is not true for signed numbers. Well, OK, it's true for the non-negative numbers in a signed representation. However, because the sign bit is the first bit in the number, adding 0's to the left of a number starting with a 1 will make it a positive number instead. In two's complement, the numbers 110 and 0110 are very different: the first is -2 and the second is 6.

So maybe we keep the most significant bit intact and pad the intervening bits with 0's? This may seem reasonable. Maybe we extend 110 by making it 100010? Just check out the math to see that this idea, while not unreasonable at first glance, doesn't hold up: the first number is -2 while the second is -30. What happens is when we shift the sign bit we also add magnitude to the number, because our sign bit in the place-value based two's complement representation is not just a visual cue as it is in sign-magnitude.

But we *do, in fact*, have to keep the most significant bit a 1. So it will *have* to represent a larger magnitude. What we need to do is instead of keeping the intervening bits 0, actually give them weight—that is, set them to 1—so they can counteract the extra magnitude being granted to the newly positioned sign bit.

What's wrong with our extension of 110 to 100010? We're off by a magnitude of 28 (-2 vs -30.) If we make the next most significant bit a 1 we reclaim 16 of that difference: the number 110010 is -14. Progress! Let's make the next bit a 1 as well: 111010 is -6. Even closer! Only four more to go! What a coincidence, the next bit represents $2^2 = 4$ so by making it a 1 as well we end up with $111110 = -2$.

Wow, this is great. It turns out that the exact value needed to offset the extra magnitude contributed to the number by an extended sign bit is that gained by making *all intervening bits 1*. So, to represent a negative number using more bits we

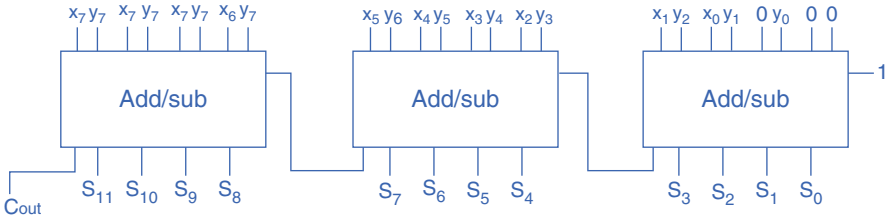


Fig. 8.11 Adder/Subtractor unit with sign extension

add enough 1's to the front of the number to make up the difference. I know, I know, I didn't believe it either when I first learned it. Go ahead and do some computations to verify:

$$1011 = -8 + 3 = -5$$
$$11011 = -16 + 8 + 3 = 16 + 11 = -5$$
$$111011 = -32 + 16 + 8 + 3 = -32 + 16 + 11 = -32 + 27 = -5$$
$$1111011 = -64 + 32 + 16 + 8 + 3 = -64 + 48 + 11 = -64 + 59 = -5$$

Since we add 0's to positive numbers and we add 1's to negative numbers, we call the process of representing a signed number using more bits **sign extension**.

Properly *sign extending* our numbers from the previous example leaves us with the adder/subtractor circuit seen in Fig. 8.11.

The leftmost bits send into this 12 bit adder/subtractor are simply the most significant bit of the numbers, x_7 and y_7. That's sign extension.

Signed Multiplication

The array multiplier from the previous chapter on unsigned arithmetic is a good starting point for our investigation into two's complement multiplication. Remember, the reason the array multiplier works is due to place-value representation and the entire point to using two's complement is that it retains that feature in a signed notation.

We have to keep in mind, however, the singular innovation: the sign bit is, in fact, negative even though it carries a weight. Working through how this affects multiplication is a great way to ensure you really grok two's complement. This is an important section for that alone. We'll work on more efficient multiplication schemes in the later Computer Arithmetic chapter, but this multiplier serves a formative role as it really helps you see the why's and how's of two's complement.

The problem we're facing is to multiply two n-bit signed numbers x and y. Since they are two's complement numbers we can write them in the following form:

$$x = sign_x + offset_x$$
$$y = sign_y + offset_y$$

The sign component is -2^{n-1} while the *offset* component is the value of the rest of the bits taken as if it were a standalone unsigned number. This is an important observation: we can use most of our knowledge from the unsigned array multiplier here. The only modification needed is an accounting for the sign component in each number.

Algebra (our familiar real-number algebra, at long last!) shows us that

$$\begin{aligned} xy &= (\ sign_x + offset_x)\Big(sign_y + offset_y\Big) \\ &= sign_x sign_y + sign_x offset_y + sign_y offset_x + offset_x offset_y \end{aligned}$$

We can see from this that the problem of multiplying our two's complement numbers has been reduced to summing these four terms. This is yet another example of how when we sit back, take careful stock of exactly what we're working with, and proceed in a precise fashion we're able to make strides and break down complex problems for solution.

How hard are these four terms to compute? The $sign_x sign_y$ term is easy: it's just equal to 0 if at least one number is positive or 2^n if both numbers are negative. The cross terms $sign_x offset_y$ and $sign_y offset_x$ are likewise straightforward: if the sign is negative it's just the two's complement of the *offset* term and if the sign is positive then it's just zero. Finally, the last term $offset_x offset_y$ is exactly the problem already solved with the array multiplier. We don't need to solve it again! We just reuse it. (In this way, we can see that digital design is no different in its abstraction than high level programming where code reuse and modularity are key organizing principles.)

Let's look at an example.

$$\begin{array}{rl} 1011 & -5 \\ \underline{\text{x}1101} & -3 \end{array}$$

Let's compute the components. First we see that.

$$sign_x = 1000 \quad offset_x = 0011$$
$$sign_y = 1000 \quad offset_y = 0101$$

It's worth stopping to notice that we maintain the *n*-bit notation for both sign and *offset* terms. Also, all four are still *signed numbers*. Therefore, the *offset* terms need to have an intact leading 0 because they are, indeed, positive. With this established we can compute the individual partial products we need.

The first term is easy:

$$sign_x sign_y = 1000 \times 1000 = 01000000$$

Again, the leading 0 is important because this product is positive. The next two terms require us to compute the two's complement of the *offset* terms and then shift them by three places:

$$sign_x offset_y = 1000 \times 0101 = 01000 \times 1011 = 1011000$$
$$sign_y offset_x = 1000 \times 0011 = 01000 \times 1101 = 1101000$$

As expected, both of these values are negative. The last term, $offset_x offset_y$, is computed via the standard method established last chapter for the multiplication of unsigned numbers:

$$offset_x offset_y = 0011 \times 0101 = 0011 + 001100$$

The second term is shifted two to the left because it represents the partial product produced by the 4's place bit in the multiplier 0101.

Now all that remains is to sum all these values together. Care must be taken to extend all the signs correctly. Here are the partial products:

$sign_x sign_y$	01000000
$sign_x offset_y$	11011000
$sign_y offset_x$	11101000
$offset_x offset_y$	00000011
	00001100
	00001111

Notice that the sum keeps carrying the 1 forever to the left no matter how far out you extend the signs. At the end of the day you end up with +15, which is the product we're after. Make sure you can work through this and follow these steps. Individually, there is nothing complicated here, but, taken as a whole, this is a wild zoo habitat filled with algebraic description, arithemetic precision, and a good dose of computing expertise required. Being able to work through these two's complement multiplications ensures you're good to go on this material.

Ten's Complement

To finish this chapter, let's look again at our family base 10 numbers. It's important to understand that all these machinations are grounded in the very reality of how numbers themselves work. It's not just a computer thing. It's an essential quality of nature that this is how quantity simply *is*. To see this, let's examine what we've done from a new perspective: 10's complement.

Let's say we want to fix the problem of using an extra visual cue to represent negative numbers in base 10. We want to apply the same concept we've just used for binary for our beloved decimal numbers.

Where would the negative numbers be? Well, we took the top half of our unsigned binary table and made those negative. Let's do the same with decimal. Limiting ourselves to 2-bit numbers, we'll say that the digits 50–99 are negative numbers. Which ones? How do we compute the 10's complement of a number? The same way as before. Subtract from 10^2. So, we end up with 100–28 = 72, so 72 is our 10's complement representation for the number − 28. We can also "flip the bits" and add one, as long as we recast "flipping the bit" as subtracting from 9: 28 "flipped" is 71, then we add the extra 1 and get 72. Yes, that's amazing.

But it's not over yet. Let's subtract 41–28 would normally require borrowing if we use the grade-school algorithm. But let's first compute the 10's complement of 28, just as we do with our two's complement adder/subtractor. This gives us 41 + 72 which is 113. That seems high until we remember the leading 1 is the *end carry* and in signed arithmetic this end carry is discarded. The remaining number is 13, which is indeed the difference we're looking for.

Wow! Seriously, work through examples like this to convince yourself of two things. First, it will help you better appreciate two's complement and see that what we are doing in the computer is not wildly different from anything we're capable of doing by hand, it's just that some peculiarities about how computers work (such as not being able to take visual cues for the sign) make this representation favorable. Second, if you are ever in charge of helping small children learn to subtract, show them this method and embrace the super fun parent-teacher conferences that will ensue. Take it, new math!

Exercises

8.1 Find the sign-magnitude and 2's complement representations of the following decimal numbers: (a) 536 (b) −212 (c) −72 (d) −12.70

8.2 Find the 2's complement of the following signed binary numbers: (a) 110001010, (b) 011010101, (c) 111.0011, (d) 10101.00101

8.3 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers: (a) 0101 (b) 1010

8.4 Perform the following two's complement additions. Indicate for each whether there is a carry out and whether overflow has occurred.

11010100	10001110	01101011	10011001
+01001110	+11110010	+01011010	+ 00110111

8.5 Write the truth table for a device which will take in a 3-bit signed binary number and output the square of that number.

8.6 Write the truth table for a device which will calculate the 4-bit two's complement of an unsigned 3-bit input.

8.7 Suppose we have A = 11010 and B = 01101.

(a) If A is in unsigned binary notation, what is its decimal equivalent?
(b) If A is in two's complement notation, what is its decimal equivalent?
(c) If A is in sign-magnitude notation, what is its decimal equivalent?
(d) Assume A and B are two's complement numbers. Calculate A + B and indicate whether overflow has occurred.
(e) Assume A and B are two's complement numbers. Calculate A – B and indicate whether overflow has occurred.

8.8 Use FA's to design a device which will calculate $9x - 2y$ where x and y are 4-bit signed inputs. Use 8-bit arithmetic.

8.9 Use FA's to design a device which will calculate $5x + 4y$ where x and y are 4-bit unsigned inputs. Use 8-bit arithmetic.

8.10 Use FA's and logic gates to design a device which will multiply a given 4-bit signed input x by -2. (Hint: if you think about this for a bit, you should see the answer is actually pretty straightforward. . .)

8.11 Use FA's and logic gates to design a 4-bit two's complement adder which has, in addition to the normal sum and carry outputs, a third output called overflow (OV). The OV output should be high when overflow occurs and low when it does not.

8.12 Let A = 1110 and B = 0101.

(a) Assume A and B are sign-magnitude numbers and answer the following three questions:

(1) What are their decimal equivalents?
(2) What are their 8-bit sign-magnitude representations?
(3) Calculate A – B using the 8-bit representations. Show your steps in sign-magnitude notation.

(b) Assume A and B are two's complement numbers and answer the following three questions:

(1) What are their decimal equivalents?
(2) What are their 8-bit two's complement representations?
(3) Suppose the input to a function F is in 4-bit two's complement format.

8.13 Define F as follows:

F = x if either adding or subtracting the two's complement number 0011 to the input will cause overflow
F = 1 if adding the two's complement number 0011 to the input results in a positive even number
F = 0 otherwise

Fill in the truth table for F and use a K-map to find the minimal SOP form of F.

8.14 Write the decimal number 17 in 8-bit two's complement notation.

8.15 Write the decimal number − 24 in 8-bit two's complement notation.

8.16 Work the following 4-bit two's complement addition problems and indicate whether each results in overflow.

$$\begin{array}{rr} 0110 & 1001 \\ \underline{+1110} & \underline{+1101} \end{array}$$

8.17 Write the 8-bit sign-magnitude and two's complement representations for each of the following decimal numbers: 115, −49, −100

8.18 Write 14 and −4 as sign-magnitude numbers and add them together.

8.19 Write 14 and −4 as two's complement numbers and add them together. Show 8-bit representations for each number and for the sum.

8.20 Fully and clearly state the overflow condition for two's complement addition.

8.21 Write 124, 85, 481, −124, −85, and −481 as two's complement numbers.

8.22 What are the largest and smallest (most negative) 16-bit binary numbers that can be represented in each of the following formats? Give you answers in decimal.

(a) unsigned binary
(b) sign-magnitude
(c) two's complement

8.23 Find the 2's complement representation of the following decimal numbers:

(a) 536 (b) −212 (c) −72 (d) −12.70

8.24 Find the 2's complement of the following signed binary numbers:

(a) 110001010
(b) 011010101
(c) 111.0011
(d) 10101.00101

8.25 Perform the following two's complement additions. Indicate for each whether there is a carry out and whether overflow has occurred.

(a) 11010100 + 01001110
(b) 10001110 + 11110010
(c) 01101011 + 01011010
(d) 10011001 + 00110111

8.26 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers:

(a) 0101
(b) 1010

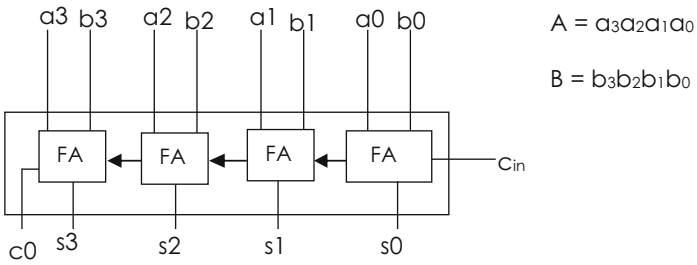


Fig. 8.12 Ripple carry adder for exercise 8.32

8.27 Use FA's and logic gates to build a device which will calculate $9x - 4y$, where x and y are 4-bit signed inputs.

8.28 Use FA's and logic gates to design a device which will multiply a given 4-bit signed input x by -2.

8.29 Find the two's complement of 101110010101.

8.30 Use FA's and logic gates to build a device which will calculate $8x - 3y$, where x and y are 4-bit unsigned inputs.

8.31 Use full adders (FA) and any combinational logic gates you need to build a device which will calculate $3x - 4y$. Assume xx and y are 4-bit signed inputs.

8.32 Using the ripple carry 4-bit parallel adder (A + B) circuit in Fig. 8.12, answer the following question.
If $X = x_3x_2x_1x_0$, use the 4-bit adder module above to draw a module that computes -4X using an 8-bit word in 2 s complement format.

8.33 Find the 2's complement of the following hex numbers:

(a) A78H
(b) 55B2H
(c) 9CH

8.34 Find the 2's complement of the following signed binary numbers:

(a) 110001010
(b) 011010101
(c) 111.0011
(d) 10101.00101

8.35 Use FA's (and any combinational logic gates you need) to build a 4-bit adder-subtracter device. It should take two 4-bit signed inputs, x and y, and a 1-bit control signal. If the control is 1, the device should output $x - y$. If the control is 0, the device should output $x + y$.

8.36 Label the six missing quantities in Fig. 8.13.

8.37 What are the largest and smallest (most negative) 8-bit binary numbers that can be represented in each of the following formats? Give you answers in decimal.

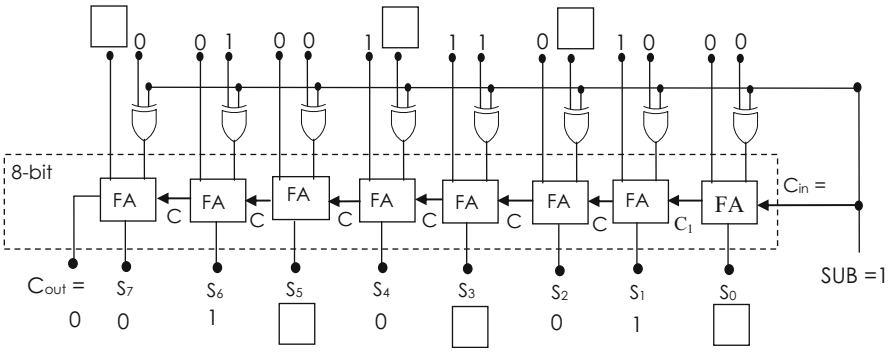


Fig. 8.13 Adder circuit for exercise 8.36

(a) unsigned binary
(b) sign-magnitude
(c) two's complement

8.38 Fill in the truth table for a device which will take a 4-bit signed input, calculate $-\frac{1}{2}$ of the input if the number is even and subtract 5 from the input if it is odd. Use 5-bit two's complement representation for the output.

Chapter 9
Other Digital Representations

Unsigned binary and two's complement are the dominant ways we typically represent numbers inside our computers. It's good to know that but important, too, to keep in mind exactly why: it's a reflection of the state of the hardware that makes them the best choices for the moment. If you know this, and know some other ways to represent information as logic-1′s and logic-0′s, you can be prepared to design new representations to fit specific needs or to respond to a change in the underlying technological environment. Yes, at the end of the day, all the ways data of any sort is encoded as 1′s and 0′s is part and parcel of the **design** of the computer. These are not given from upon high—they are all within our ability to choose and reject based on the parameters of a given application.

What follows in this chapter is a selection of numeric representations that have special, even niche, application. Some of them you may draw upon as needed when the application is right, and others may prove an inspiration for a particular representation you need to design yourself.

Binary Coded Decimal

The first take on arithmetic in the history of digital electronic computing was a representation called **binary coded decimal**, or BCD. Here are examples of BCD encodings:

$$345 = 0011\ 0100\ 0101$$
$$962 = 1001\ 0110\ 0010$$

The idea is to translate each decimal digit directly as a binary number and then string together all the 4-bit representations of each digit into a coherent whole. This representation is not as **compact** as unsigned binary—it would take only 9 unsigned binary bits to represent 962 while BCD requires 12. It also leads to some issues with addition.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_9

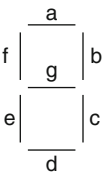
Fig. 9.1 Unsigned binary addition

```
 0010 0100   24
+0011 1000   38
 0101 1100
```

Fig. 9.2 Correcting binary coded decimal arithmetic

```
 0010 0100   24
+0011 1000   38
 0101 1100
 0000 0110   +6
 0110 0010   62
```

Fig. 9.3 Seven segment display



Say we want to add 24 and 38, as in Fig. 9.1

This gives the strange number 512, where the 12 is one digit? What? It's not even translatable into decimal. It could mean 50 + 12 which is the correct answer, but what has happened is that the addition has knocked our numbers outside of the representation. We need to fix this.

The issue is that BCD only uses the digits 0–9 which, in a four-bit representation, leaves the six bit strings 1010–1111 unused. Since our normal binary addition algorithm doesn't "know" we are in BCD it doesn't "know" to jump past those last six spots and therefore we may end up there and result in a number that doesn't make a lot of sense.

Fortunately, there's an easy solution: if we add 6 to the offending digit, we get back to where we need to be.

So, continuing our example, we have Fig. 9.2.

For larger numbers this check needs to be made on each digit (each group of four bits.) Because of all this checking and adding six over and over again, BCD numbers have fallen out of favor as the representation of choice for high speed arithmetic. But what they are good for is when it's to our advantage to work with decimal numbers directly.

This comes up when we're displaying decimal numbers in, say, a digital clock or timer. Suppose we want a countdown timer to display the decimal number 500 and then count down to 0. We need to display each of three different digits. A typical display type is called the **seven-segment display** (Fig. 9.3.)

In such a display you have seven output logic functions which drive the individual segments a through g. Each digit lights up different segmens. For example, 1 activates only b and c, 2 activates a, b, g, e, and d, while 8 activates all of them.

Suppose we encode our count in unsigned binary. This requires 9 bits. We want to display on three seven-segment displays. That is a total of 21 output functions. To specify the functions would require a truth table with 9 inputs (512 rows) and 21 columns for a grand total of 10,752 entries we'd have to detail.

Let's try a different approach. What if we instead encode our number with three BCD digits (total of 12 bits.) Now it appears we worse off because we'd have a truth table with 4096 rows and a grand total of 86,016 table entries! There goes the weekend!

But, if we're clever we can see that all we really have to do is map a **single decimal digit representation** to a **single seven-segment display**. That is, if we have a device that can drive a seven-segment display for a single 4-bit BCD digit then we can just network three of those together to control the 500-to-1 countdown.

This efficiency is entirely because we are using a decimal encoding to display a decimal number—no translation from binary is necessary. In fact, we only ever have to design the device once, because if we increase the digits needed in the display all we ever have to do is add a few more devices to the network. Compare to if we used a binary number and then we'd have to re-specify 7 new output functions per digit in a table with 2^n rows, where n is the number of bits we'd need. It's not at all efficient.

This point is really important to grasp: the way we use the number affects the way we need to encode it. It's not as simple as saying "it's all 1′s and 0′s" or even "it's all binary" or even "it's all two's complement."

Decimal Codes

There are other ways to encode decimal numbers directly without using the unsigned binary approach. **Weighted decimal codes** apply place-value within each four-bit digit, and they can do it in different ways.

Our BCD code weights the bits 8421: that is, the most significant bit has weight $2^3 = 8$, the next bit has weight $2^2 = 4$, and so as as we would expect. These are not the only weightings that can work, however.

One can apply a 5421 code where the most significant bit is now weighted as 5, the next bit as 4, etc. In this system the number 8 would be represented as $1011 = 5 + 2 + 1 = 8$.

Another option is the 2421 code where 8 would be written as $1110 = 2 + 4 + 2 = 8$. One advantage of this code is that it is **self-complementing** which means you can find the nine's complement (9 – the number) by simply complementing all the individual bits. With $8 = 1110$ we get that $9–8 = 1 = 0001$ which we get by "flipping" all the bits of the representation for 8. Also, $5 = 1011$ while $9–5 = 4 = 0100$, again found by complementing all the bits (this also gives us the reason why we'd choose 1011 for 5 instead of 0101, the other option compatible with the weighting.) Self-complementing representations are useful in various error-detection and security applications.

Table 9.1 Decimal codes

Decimal	BCD	XS3	5421	2421	7536	2-out-of-5
0	0000	0011	0000	0000	0000	11000
1	0001	0100	0001	0001	1001	00011
2	0010	0101	0010	0010	0111	00101
3	0011	0110	0011	0011	0010	00110
4	0100	0111	0100	0100	1011	01001
5	0101	1000	0101	1011	0100	01010
6	0110	1001	0110	1100	1101	01100
7	0111	1010	0111	1101	1000	10001
8	1000	1011	1011	1110	0110	10010
9	1001	1100	1100	1111	1111	10100

We can even get crazy and have negative weightings. The $753\bar{6}$ code assigns the weight of -6 to the least significant bit! Now we have $8 = 0110$ and $4 = 1011$. This is also a self-complementing code.

There are important **nonweighted decimal codes** as well. The Excess-3 code (XS3) is exactly the BCD code shifted by 3 numbers: 0 is 0011, 1 is 0100, 8 is 1011, and 9 is 1100. Its advantage over BCD is that it is self-complementing.

Another nonweighted code in wide use is the 2-out-of-5 code in which each digit is represented by a 5-bit string in which exactly 2 of the bits are 1. This is used in bar codes and is great for optical scanner to pick up and has excellent error detection and correction capabilities.

All these representations are summarized in Table 9.1.

Gray Code

Decimal encodings are useful for displaying numbers. Another family of representations, called **unit-distance codes**, are good for labeling things. In our normal binary progression, to go from 3 to 4, that is from 011 to 100 requires changing all three bits. In a unit-distance code we carefully select the progression so that exactly one bit changes each time.

The most famous unit distance code is the gray code (Table 9.2).

You've already seen the utility of this code as it forms the basis of how we organize the cells in our K-maps. We will also see how helpful it can be as labels for the states in our state machine designs we'll get to in later chapters. It's used in other areas of computer science as well, such as in genetic algorithms.

Alphanumeric Codes

In principle, you can choose any combination of 1′s and 0′s to correspond to any other sort of character you may need in computing: letters, punctuation, and special inputs such as the carriage return or resetting of the island's doomsday clock from

Table 9.2 3- and 4- bit Gray codes

	Gray code	
	3-bit	4-bit
0	000	0000
1	001	0001
2	011	0011
3	010	0010
4	110	0110
5	111	0111
6	101	0101
7	100	0100
8		1100
9		1101
10		1111
11		1110
12		1010
13		1011
14		1001
15		1000

hitting ENTER. In practice, it's helpful to have a standard way of doing this so that computers can communicate with each other using the code. The **American Standard Code for Information Interchange (ASCII)** is a 7-bit code for doing just this. ASCII table are ubiquitous and easy to find simply by searching. Any programmers using character data types would do well to familiarize themselves with the notation.

We also have the 16-bit Unicode which includes many more characters than are needed in American English. It's important when working on programs involving character data to know exactly how the values are being encoded. While ASCII used to be the universal standard, and is still quite common in microcontroller work (especially the 8-bit ones), Unicode has taken over much of the PC market. When you declare characters these days, chances are it's a Unicode encoding you'll be working with.

Fixed Point Numbers

A **fixed-point number** is one where the *radix point* (the general name for the decimal point in base 10 or the binary point in base 2) is in a *fixed* location. Our unsigned and signed binary representations are fixed-point systems with the radix point to the right of the least significant digit. That's not very interesting. Typically, we use fixed-point representations when we need some fractional precision but we know ahead of time exactly how many bits we want to use to represent it. We call the number of fractional bits the *resolution* of the representation.

Floating-Point Representations

In contrast to fixed-point numbers, in a **floating-point number** the location of the radix point is, well, not fixed: it is allowed to *float*. We usually encounter these numbers in a beginning programming course when we learn to declare `float` variables. In that context, we are usually trying to represent fractions such as 4.50 and the like. We tend to then conflate the concept of a floating point number with a fraction. But, as we just saw in the section on fixed-point numbers, we don't have to use a floating point representation to encode fractional numbers. It's important to free ourselves from this way of thinking and to see that floating point numbers are more powerful than that. The fact that the radix point is movable means that it can also help us encode numbers larger than those we could otherwise get a handle on. Remember our scientific notation from high school science class: numbers like 4.71×10^{30}. These are also floating point numbers.

We're going to need fields to capture the salient features of our floating point numbers. The key fields for us are **sign**, **integer**, **fraction**, and **exponent**. The fraction field is often called the *significand* field.

For our decimal example 4.71×10^{30}, we have that the *sign* is positive, the *integer* is 4, the *fraction* or *significand* is 71, and the *exponent is* 30.

In our binary examples, we'll be looking at numbers of the form 1101.0110×2^{19}. In this example, we take the *sign* to again be positive, the *integer* to be 1101, the *fraction* or *significand* to be 0110, and the *exponent* to be 19.

Because we have a certain number of bits to encode the entire number, and interesting dynamic obtains: the more bits we allocate to one field the fewer the amount of bits we have left to supply the other fields. This creates a tension within the encoding as we have to decide what to prioritize and then run with it.

For example, in a 32-bit encoding, we could assign 1 bit to the sign, 2 bits to the integer, 27 bits to the fraction and 2 bits to the exponent. This gives us the ability to represent numbers like $11.00110101000111010101001110 1 \times 2^2$ but no ability at all to represent numbers larger than 4 (the supremum of the integer + fractional part) $\times 2^3$ (since we only have two bits for the exponent we max out at 3) $= 32$. Comparatively, in a 32-bit integer encoding we can work with numbers like 4 billion. So, it's worth taking some time to look at these tradeoffs. In this case, we gained a lot of new fractions we can represent at the cost of not being able to represent a lot of integers. We still have the same number of 2^n total bit strings available to us; the only difference is which 2^n numbers do we assign these bit strings to.

We formalize this tradeoff when we speak of a floating point representations' **range** vs. its **precision**. The *range* marks the difference in magnitude between the largest and smallest numbers expressible in the representation. The *precision* tells us the depth of fractional expressive power of the representation. Since we only have so many bits available to us, every bit devoted to range is a bit taken from precision and visa versa. Because of this, and continuing our theme in this chapter that the encodings we use to represent our information inside a digital system is part

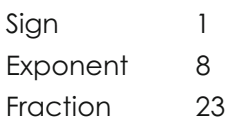


Fig. 9.4 IEEE single precision floating point format

of the actual design of that system, we can see that the precise allocation of bits between range and precision needs to be well-thought out based on the *needs* the format is addressing.

While many specialized representations exist for various scientific and control uses, in order for computers to communicate with each other effectively a standard format needed to exist. The Institute for Electronics and Electrical Engineers, **IEEE**, (pronounced "Eye-Triple-E", not as if you were screaming) promulgated just that. The IEEE standard formats are the ones used by default by the vast majority of computers out there. For that reason, we'll discuss them as our core example.

In the **32-bit IEEE single-precision floating point format** (Fig. 9.4) we allocation 1 bit to the sign, 8 to the exponent, and 23 to the fraction.

The first thing to notice is that there is a dedicated sign bit. That's right—this format is indeed in sign-magnitude representation rather than two's complement. This works out because the floating point format is laden with multiple fields anyway and from the beginning has no hope of being a place-value representation. Remember, the downside to sign-magnitude was the preprocessing required to perform arithmetic. Here, we're going to require a lot of preprocessing regardless, so having a stand-alone sign bit is fine.

The second thing to notice is that there is *no integer field at all!* Did they forget? How can they devote zero bits to the number before the radix point? The key to understanding what is going on here is to note that in IEEE standard formats the numbers are always represented in a **normal form** (just like with logic functions, other mathematical objects also have normal forms). The normal form we care about here is one where there is exactly one bit before the radix point.

For example, the decimal number 18.75 can be written in binary as 10,010.11. This is not the normal form, however. To put 10,010.11 into normal form we need to move the radix point and write it as 1.001011×2^4. To summarize:

Decimal number: 18.75
Binary equivalent: 10010.11
Normal form: 1.001011×2^4

We can now see why we don't need a field devoted to the integer portion of the number. By limiting the space to the left of the radix point to a single bit, we're assured that bit is always 1 (for a nonzero number.) Since it's always 1, we don't even need to encode it. We'll just build the hardware that interprets the number in such a way that it always inserts the 1 in the correct location. This is sometimes called a *hidden bit*.

But, what about the number 0? It's the only number that will not have a 1 in that bit position. Is the IEEE format incapable of representing 0?

It turns out that floating point formats typically have a lot of edge cases. They have to represent not only overflow in which the result of a computation lie outside the expressible range because the number is too large in magnitude but also situations where the result of a computation lies outside the expressible range because the number, while not too large in magnitude, nevertheless requires too much precision. This is just like an integer representation can't handle 5/2 not because 2.5 is too large or too small but because the integer representation has no capacity for dealing with the .5.

So, floating point formats often have special *reserved codes* within them to signal the interpreting hardware of a variety of special conditions. In the IEEE standard formats these reserved codes reside in the exponent field. Instead of using all $2^8 = 256$ possible 8-bit encodings for actual exponent values, the smallest (all zeroes) and highest (all 1′s) are reserved for special situations. One of those is, indeed, the representation of zero. If the exponent field is all 0′s, and all the fractional field bits are 0, then the number is, zero. The standard mandates that we try our best to use the positive sign with the zero, but like in any sign-magnitude notation it is possible to have two representations of zero, one with a positive sign bit and one with a negative sign bit.

There is one other tweak to the regular interpretation of fields included in the IEEE standard format. Consider how the number is laid out: sign bit, then exponent field, then fraction field. While the fraction field is always an unsigned number, we do need to represent both positive and negative exponents. Therefore, the exponent field must be written in a signed notation, in this case two's complement.

That's fine, but we end up with an awkward situation when it comes to the design of the hardware that processes these numbers. The way two's complement works, the numbers large in magnitude are the negative ones, and negative exponents represent very small numbers. If we are trying to perform magnitude comparisons on floating point numbers, we strip out the sign bit and then see what remains: an signed exponent field followed by an unsigned fraction field. Combining these, our magnitude comparator is designed to work with unsigned numbers. It's complicated that the large-in-magnitude exponents actually represent the smallest numbers. We'd like to map the negative exponents to numbers that are actually small in magnitude. To do this, we **bias the exponent**.

We want to, in effect, swap the sign bit of our two's complement exponents. We want the negative exponents, which normally lead off with a 1, to reside in the space of encodings with a leading 0. To do this, we have to add 01111111 to the exponent. This is a case where we find working in hexadecimal directly to be helpful, and express the bias as `x7F`.

Let's work through a full example and see how this process all comes together.

Consider the decimal number 82.625. This can be represented in binary as 1010010.101 (make sure you can do this conversion.) Putting it into normal form we have 1.010010101×2^6. The sign bit is 1 because it's a positive number. The fraction field is 0100101010. . .0 (total 23 bits). Remember we do not encode the

1 to the left of the radix point! Now, the exponent: we start with 6 and add the bias of `x7F` and end up with `x85` = 10000101 for the exponent field. The final number then is 01000010101001010 10. . .0 (total of 32 bits). It's often convenient to write this in hexadecimal, which in this case would be `x42A54000`.

To summarize this example:

Decimal number:	82.625
Binary:	1010010.101
Normal form:	1.010010101×2^6
Sign bit:	0
Exponent field:	6 + x7F = x85 = 10000101
Fraction field:	01001010100000000000000
Final number:	01000010101001010100000000000000 = x42A54000

There is also a **64-bit IEEE double-precision floating point format** which has a 12 bit exponent field which uses a bias of `x7FF` and a 51 bit fraction field. This format is also commonly used in computing.

Designing an Encoding: Days of the Year

An important takeaway from this chapter is that the way we represent data inside the computer is **up to you as the designer!** Everything is related: from the way we ascribe meaning to the logic-1 and logic-0 signals our electronics produce to the way we connect our digital components to the types of instructions our processors can use for computations. This example will showcase the thought pattern behind designing our own representation and help us grok the idea that we have to put some creative energy into doing so.

Suppose we want to encode the days of the year. There are many ways we may want to use this encoding. Here are some possible questions we may need to ask of two days X and Y:

Q1: Is day X earlier or later in the year than day Y?
Q2: Is day X in the same month as day Y?
Q3: Is day X earlier in the week than day Y?

To handle Q1, we can simply number the days 0 through 363 and convert to the normal 9-bit unsigned binary representation. This is the most compact way to design this encoding, but it's utility is limited. We can answer Q1 since the comparison hardware will just compare the magnitudes of the numbers and everything will work out, but we can't efficiently respond to more involved uses of the data.

For Q2, for example, we need more information than what that encoding provides. Yes, we could design a complex device that can input a number in our Q1 format and decode what month it is in (not quite as easy as just dividing by 12) and then perform the comparison. But, it's much more efficient for our overall system

design to instead break our encoding of the day into fields. We can have a 4-bit field for the month itself and then a 5-bit field for the day. As our previous design, this also requires 9 bits. The key is that now we can use the encoding in different ways. If we want to answer Q1 with this encoding, we have a harder time: we must first compare the month fields and then compare the day fields. The previous encoing was more efficient for Q1 and it was hard to answer Q2 with it while this encoding is more efficient for Q2 than it is for Q1. It's really important to get the ebb and flow of this here. Again, the way we represent all our information within digital system is an important part of the overall system's design and you should always be on the lookout for efficiencies to be gained in this way. Often stock representations will be fine, especially for routine calculations in area where standards exist (such as IEEE floating point standard), but for specialized needs the task is on the designer to come through.

To answer Q3, we need to augment either of our previous encodings with a 3-bit week field. Now we're looking at a 12-bit encoding. This may seem less efficient but keep in mind that the number of bits used to represent our data is not the only metric we care about. We need to compare this lack of compactness in the bit representation with the overall complexity of the system that needs to process the representation. Pulling out weeks from one of our previous two formats would require a very expensive specialized device. If the day of the week is an important use of this representation then it behooves us to make the tradeoff of more bits for simpler system hardware.

Exercises

9.1. Design a 3-bit Gray Code encoder. Your solution should include a completed truth table, algebraic output equations for each output bit, and a combinational logic diagram implementing those output equations.

9.2. Write 17 and 29 using both the BCD and XS3 encodings.

9.3. Design a device that will take as input a number from 0 to 9, encoded in XS3 notation, and return a 1 if the number is prime. Show the truth table for this device and use a Karnaugh map to find the simplified SOP form. Draw the gate-level logic implementation for this design.

9.4. Design a BCD encoder. It should have ten input lines (I_0 through I_9) and give a four-bit output corresponding to the BCD representation of the input line's decimal equivalent. Also include an Enable input line. The device should output all zeroes when the Enable input is 0, and it should output the correct encoding when the Enable input is 1.

9.5. Design an XS3 decoder. It should be able to handle a four-bit XS3 encoded input, have an Enable input line, and have ten output lines (corresponding to the decimal numbers 0 through 9.) The output line asserted should be determined by the XS3 representation detected on the input lines. If the Enable input line is asserted, then the device should output the correct decimal

decoding; if the Enable input line is 0, then the device should not assert any of the output lines.

9.6. Design a controller for the *d* segment in a seven-segment display. Assume BCD inputs and show the gate-level implementation for the minimal SOP form.

9.7. Design a device that will convert from BCD to XS3 or from XS3 to BCD, depending on a control input signal. Your input should consists of a 4 bit number and a 1 bit control signal set to 0 if converting from BCD to XS3 and to 1 if converting from XS3 to BCD. Draw the implementation diagram for this device. You may use Full Adders as blocks in your design.

9.8. Write the following in IEEE single precision floating point format:

(a) −34.5
(b) 127.90
(c) −7.593,412

9.9. Design a digital representation of colors using 3-bits. In a subtractive system we begin with white and filter with the primary colors (red, yellow, and blue). If we represent white with 000 our representation system ought to be able to produce the other colors based on the appropriate computation: red + yellow = orange, for example. Your design should include white (000), black (111), and the primary (red, yellow, blue) and secondary (green, purple, orange) colors.

Chapter 10
Encoding Code

We're using the computer as the main motivating example throughout our development of digital design, so let's get to programming!

We discussed at the outset that the core characteristic about our computers that we cannot undervalue is that they are **programmable**. That is, they can be configured to perform a variety of computations. We are familiar with programming computers in languages such as BASIC, C++, Java, Python, and Ada. The question then becomes how do we get the code inside the computer? We know that everything in our digital electronic computer must be represented as strings of logic-1's and logic-0's. We've just gone through three chapters illustrating myriad ways to use these two values to represent numbers. But how can we represent code this way? We can just divide a decimal number by two over and over again to get its binary representation. Can we divide a for loop or a function call by two over and over again? That doesn't sound sensible. There must be another way.

Instructions and Datapaths

First of all, we need to figure out what sort of code—what programming language—the computer can work with internally. It can't keep up with the proliferation of new languages out there. You may recall from your programming courses that the languages we write in have to be compiled, or translated, into a certain kind of language, called **assembly**, that computer can understand. The reason for that is now evident: if we are to design a digital circuit to implement computer programs the languages it can handle must be fixed at the time of construction. Since we want our computer to be able to support programming languages beyond those available when it's built, it makes sense to give it its own personal language to then have the others translated into. Also, this gives us as computer designers control over the programming of the language. We don't have to try to support every crazy option out there in a direct way. We can focus one by one on the statements, or

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_10

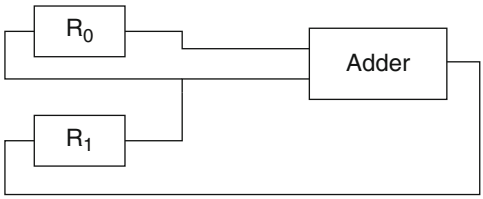


Fig. 10.1 Basic datapath

instructions as they are called in assembly languages, and optimize our design to run those instructions efficiently.

So, when we talk about how to encode our code in logic-1's and logic-0's, we are able to step back from C and Cojure statements and consider only those operations which relate directly to our circuit.

What kinds of things are we going to want to support? The main elements of imperative programming are variables, structured data such as arrays, loops, conditionals such as if statements, and function calls. If we can do all of those things, then we're going to go home happy and call it a day.

Just as declaring and assigning values to variables is at the heart of our imperative programming, the digital component called the register (we'll see the design of these next chapter) and information transfer among registers is at the heart of our digital systems. All our instructions really do is move a value from one register to another, going through some other devices along the way. The nature of those devices determines the **operation** performed and the specific registers used are called the **operands**.

For example, consider the circuit given in Fig. 10.1.

This is called a **datapath** for a computation. Each datapath can implement the basic instructions the computer is able to carry out. The elements of a datpath are the block diagrams for each of the components as well as the wires that connect them. Everything in this diagram is physical and we are limited in our programming of this computer by what physical devices and connections are available to us.

The design can support an instruction like ADD R1, R0 which adds R1 to R0 and stores the sum back in R1. We can see this because the Adder has inputs from both R0 and R1 and its output is connected to R1's input. Therefore all the conditions for ADD R1, R0 are supported. We call ADD the operation and R1 and R0 are operands.

We can also support a MOV R0, R1 operation which copies the value from R1 into R0. We know this because there is a physical wire connecting R1 to R0. We can see it in the diagram and follow the flow of information in the datapath. In that case, MOV is the operation and R1 and R0 are again the operands. R1 is called the source operand and R0 is the destination operand. We're going to adopt the convention that the first operand in the list is the destination and the others will be source operands. (The first may double as a source operand as well, as we saw in the ADD R1, R0 instruction above.)

How do we represent these instructions inside our digital computer? If we focus on these key elements, operations and operands, we can see that the complexity of

the encoding problem is reduced tremendously. We can simply use our 1's and 0's as labels and label each of the operations in our language with some string, and number our registers as well to refer to them by number.

What we end up with are a set of **fields** that make up and **instruction.** The **opcode** field tells us what path the data signals are taking—what devices the signal will travel through as the instruction is executed. The **operand** fields tell which register are involved in the flow of data. Our circuit design (which is the topic of the next several chapters) will do the rest of the work.

We can see now why assembly languages are so intimately tied to a specific machine: they are a reflection of the precise wiring of a given computer. Change the numbers of registers or how the various devices are connected to each other and you change the instructions and modify the very language the computer is able to process.

Variables and Assignment

Registers and variables go hand in hand. We associate a variable with a register or a memory location, which for our purposes can be considered to be, in effect, a register outside the datapath of the computer.

Here's an example of some arithmetic instructions:

```
ADD Rdest, Rsrc1, Rsrc2    Rdest = Rsrc1 + Rsrc 2
SUB  A, Rsrc               A = A - Rsrc
MOV Rdest, Rsrc            Rdest = Rsrc
```

The ADD instruction has a single destination register and two source registers. It will therefore require enough bits to encode all three of those operands. The SUB instruction, in contrast, has only a single source operand. The Accumulator register A is a system register whose use is hardwired into the operation itself. Older machines used this design to save bit space in the instruction encodings. Modern machines don't need to save bit space and typically let the programmer (or, increasingly, the compiler) specify all the necessary operands. This is one of the ways advances in electronics technology affects the very language we use to express our use of computation to solve problems in the world. Finally, the MOV instruction requires two operand fields.

All instruction require an **opcode** field to indicate what path through the circuit is being utilized. We usually think of the opcode as the operation itself, but it can be a many-to-one relation as indicated by the following instructions:

```
MOV Rdest, Rsrc              Rdest = Rsrc
MOV Rdest, #N                Rdest = N
MOV Rdest, Addr              Rdest = M[Addr]
MOV Rdest, Rscr, Offset      Rdest = M[Rscr + Offset]
```

Now we have four different ways to encode the operand itself. All these instructions perform the same abstract operation: they assign (or *move* in assembly parlance) a value to a destination register. The path used by the source operand, however, differs in each case. Because of this each of these instructions would require a different opcode even though as a programmer we lump them all in the same category operationally.

Why do we need different ways to encode an operand? Why can't we always just use a register? Let's look at each instruction and see what it does.

The MOV Rdest, Rsrc is the simplest of the group. We call this a **register operand** because the value we are sending through the circuit is given by the register number in the operand field.

The MOV Rdest, #N instruction loads Rdest with a specific number N. If we write code such as x = 4; we don't need to store the 4 in another register first and then load that into x. We just want to treat 4 as a literal and store it in x. That's what this instruction does. We call this an **immediate operand**.

The MOV Rdest, Addr instruction loads Rdest with the contents of the memory location given by the operand Addr. This makes sense. We have very few registers in the processor and we may have a lot of variables we need to access. It stands to reason that some of these variables will be stored in memory. We call this a **direct operand**.

The MOV Rdest, Rscr, Offset instruction loads Rdest with the value in the memory location given by the sum of the value in Rscr and the Offset operand. What's going on here. We often write arrays in our high-level language and access elements through an index variable such as x[i] = 4; This is how assembly languages can support structured data. The value in Rscr acts as the index of our array. It lets us iterate through an array by incrementing Rscr (we can set Offset to 0) or, if we let Rscr be the beginning of an array we can use Offset to access a specific element. This is called a **displacement operand.**

Overwhelmed? Well, there are still many others. Taken together, the various ways we can encode operands in instructions are called **addressing modes** because we often use the world **address** to refer to the location inside the computer where a given value is stored. So the addressing mode tells us how to compute the address of the value we're after. Right now we'll only work with simple ones until we've had more experience with this.

But what is the reason for so many addressing modes? Basically, the idea is that the address is a number requiring a great many bits—often too many to easily fit within the encoding of the instruction itself. Therefore, all these addressing modes are just schemes to figure out what the addresses of operands ought to be. Each is used in accord with a specific programming construct. Displacement addressing, for example, is a drect outgrowth of the array data structure and its behavior mirrors that of array indexing.

All of these instructions assign a value to the destination register that is exactly equal to the value indicated by the source operand (once you go through all the steps to unpack exactly where that value is located in the computer.) We can also send values through digital components such as an adder or multiplier, en route to their destination. In this way we can have instruction perform arithmetic or logic operations.

Consider:

```
ADD Rdest, Rsrc1, Rsrc2    Rdest = Rsrc1 + Rsrc2
XOR Rdest, #N              Rdest = Rdest XOR N
INC Rdest                  Rdest = Rdest + 1
```

The ADD Rdest, Rsrc1, Rsrc2 instruction will use the adder to compute the sum of its source registers and put the result in Rdest. This behavior is illustrated in the simple datapath at the opening of this chapter.

The XOR Rdest, #N instruction will perform the bitwise XOR of Rdest and the immediate operand N. In this way we can selectively complement individual bits of a value.

The INC Rdest instruction will add one to, or *increment*, the value in Rdest. Why have this when we could just use an ADD instruction instead? Great question! The exact mix of instruction any given computer is going to utilize is a heavily designed one and some will include specialty instructions such as INC and others will force you to use ADD to accomplish this. Some don't even have MOV, for example, and instead have you use ADD with a 0 to simply copy a value into another location! Computers like these may be frustrating to program (or a fun challenge, depending on how you view such things) but we are increasingly equipped to understand the tradeoffs the designers of these machines are faced with. There are only so many instructions we can use in a machine so decisions must be made.

For example, some instruction sets can skimp on any instruction for subtracting because, in tandem with bitwise logic and addition, and armed with the power of two's complement representations, we can program subtraction for ourselves.

Recall that we can subtract by computing the two's complement of the subtrahend and we may do so my complementing every bit and then adding 1. Further, we complement each bit by XORing with 1's. Thus, in an instruction set featuring ADD and XOR instructions we may evaluate the subtraction A – B thusly:

```
XOR B, #0xFF      XOR each bit of B with a 1
ADD B, #1         Add 1 to B to get -B
ADD A, B          A = A + B
```

Not all instruction sets need to make such drastic cuts in basic functionality, however. A deeper discussion of these issues can be found in specialty texts on computer design. As a computing scientist it's interesting to consider what instructions are necessary to implement all the routine sorts of actions we care about when writing our high-level programs. We then need ot weigh the concern of simplicity of instruction set against the complexity required of a compiler which must translate our C++ code into these 1's and 0's. Further study into the nature of compilers will really help the interested study see these connections further. Our purposes in this chapter are satisfied with merely pointing out these considerations.

Conditionals

The previous section covers instructions capable of working with basic variable assignment and evaluation of arithmetic expressions. What about `if` statements? How can an instruction handle a statement like if (x > 4) y + 2; where y is to be updated only in the event x meets a condition? It would seem like we're looking at quite a few fields here: a variable x, a condition >4, and an entirely separate statement y + 2 as well! How can we break this down so our digital components can do something with it?

The key here is remembering what we're doing: encoding code. We are turning our program statements themselves into strings of logic-1's and logic-0's. To what end? To store inside the computer's memory, of course. But what does that mean? It means that our statements have a specific label to indicate which memory location they are in. So, we have an identifier for our statements. Wow! This is pretty new to those of you who've never programmed with labels or line numbers. We are used to identifiers being associated with variables and larger data structures, but we are definitely not accustomed to thinking about our statements as having identifiers. Within our computer, they do!

How do we tell the computer to execute an instruction at a particular location, however? It turns out we have a register called the **program counter**, **PC**, that stores the location of the next instruction to execute. Our hardware will automatically update the PC to execute the next instruction in order (thankfully, because that's exactly what we expect to happen). If we want a different instruction to execute, then, we can just change the value in the PC. We call this **branching** or **jumping** to another instruction and often use the mnemonic BR or JMP for this operation.

This helps us a great deal when it comes to encoding the if statement. All we have to know about the separate statement y + 2 is its location, not its operation or operand! We can implement it as a different instruction stored in its own location and just refer to it from the instruction implementing if.

What about the condition itself? Seems $x > 4$ is actually another statement entirely? In fact, in many languages you can put anything that evaluates to a Boolean value there. We really have the same problem as with the y + 4 statement. We basically have two options here: (1) we can leave it as a separate instruction or (2) we can severely limit what Boolean expressions we allow.

If we take route (1) then we need a place to store the result of the comparison. Luckily, we have what is called a **status register** in the datapath designed just for this purpose! We can make an instruction that compares x to 4 and then updates a status register bit if $x > 4$. Then when it comes time to execute the if instruction we can just check the status register bit to see whether we need to execute y + 4 or just move on with our lives.

If we take route (2) then we need to choose something that's easy to work with. Typically, something like $x > 0$ is chosen. If we want to check whether $x > 4$ we'd just have to check whether $x - 4 > 0$ instead. It's still on us to encode the variable

we're checking against 0, however, so our if instruction will need to have a field for that.

OK, let's put this all together.

```
BRg, N      If the g bit is set in the status register, then PC = PC + N
BR R0, N    If the value in R0 > 0, then PC = PC + N
```

Why are we adding N to the PC instead of just setting PC = N? Usually we want to branch to instructions that are located nearby the branch instruction. Since memory locations are very large numbers (memories are very big these days) it's more efficient to encode a smaller offset N and then sign extend it and add it to the large value in the system register PC than it is to figure out how to encode a full memory location directly. It's not impossible, and some computers do it this way. Just be aware that it takes more design work and if you're OK with the tradeoffs then run with it.

So, to show the full example, let's look at the statement

```
if (a = b)
    a = a + b;
else
    a = a - b;
```

We know how to independently handle the various calculations needed: a == b, a = a + b, and a = a − b. What we need to do to implement the if statement is to combine the calculations with our knowledge of the fact that lines of code have addresses. We end up with something like the following:

```
        JNEQ a, b, ELSE
        ADD a, b
        JMP EXIT
ELSE:   SUB a, b
EXIT:
```

The program continues following EXIT. In this example, ELSE and EXIT are labels—that is, stand-ins for memory addresses corresponding to the particular instructions. The instruction JNEQ will *jump if not equal*—it will add enough to the PC to make it so the next instruction Fetched is the one located at the ELSE label, provided the condition a = b is met. If that condition is false, then the PC will not be disturbed and we instead execute the next instruction in the code. That's why we must have an unconditional jump JMP EXIT as the next instruction so we bypass the ELSE clause. What in high-level code is intuitively assumed to not be executed because we grasp the flow of the program must at the machine level be meticulously prescribed as arithmetic on addresses (and thus requiring access to the PC register.)

Loops

Loops can be thought of as special case conditionals: if the loop exit condition is not yet met then stay within the loop, else exit the loop. Because of this loops work off the same principle as the conditionals: address arithmetic. The target of the loop will have an address and the loop statement itself will modify the PC in such a way as to get program control to the appropriate instruction: either that following the loop or the beginning of the loop. Let's look at an example:

```
while ( a < 10)
{
  b = b + a;
  a = a + 1
}
```

At the machine level this would look something like the following:

```
LOOP:     SUB b, a, 10
          JNEG  b, EXIT
          ADD b, b, a
          ADD a, a, 1
          JMP LOOP
EXIT:
```

The JNEG instruction will *jump if negative*: that is, it will add enough to the PC to make it so the next instruction to be fected is the one after the termination of the loop, at the label EXIT. If that condition is not met, then we'll execute the body of the loop and then hit the unconditional branch instruction JMP LOOP which takes us back to the beginning.

More complex conditional and loop programming structures can easily be generated by combinations of these concepts. The key to remember is that it's all about the manipulation of the place in the code via arithmetic involving the PC. Also, the condition itself, what we usually consider the boolean expression, must be evaluated separately and prior to the instruction that checks the condition so that an if (cond) statement turns into a several-instruction sequence that may seem ponderous. But that's as it should be! It was not by accident that we developed high-level languages! They allow us to abstract away from the datapath of the machine and think about higher things. It's a really great thing to do, if you like programming at all, to play around with this. It will give you a fantastic appreciation for the idea that abstraction in computer programming is fundamentally about *what the programmer must think about*. Sometimes it's proper to use Java or Python to develop an application or prototype some algorithm because what the programmer cares about is the formal structure and appropriate computational expression

thereof regarding the problem domain under consideration. But sometimes it's necessary to dig into C or assembly code and ensure the most efficient care has been taken with the internal components of the machine. The programming language chosen for a task must fit not only the human programmer's particular likes and dislikes but the requirements of the task at hand and a study of computer datapaths and the instructions that run on them can help aid in this understanding.

Digital Representation of Instructions

Our instructions require specification of three key elements:

1. Location of data used in the operation
2. What operation to perform
3. Location where the result should be stored

We refer to these as **source operands, operation,** and **destination operands.** When we represent our instructions digitally, we need to design our encoding such that all three of these elements are present and efficiently usable by the larger system. See Chap. 9 for details on encoding non-instruction information in similar ways.

Modern instruction set design typically revolves around breaking the encoding into fields, just like how we handled things like floating point numbers, and then assigning strings of 1's and 0's within the fields to the relevant physical system elements.

For example, one of the most critical design decisions when putting a computer together is the number of general-purpose registers we want to use for operands. This choice directly affects the encoding space within an instruction.

With a fixed number of bits available for an instruction, we end up forced to make tradeoffs. Every bit spent on the opcode field is a bit we cannot spend on a register or immediate operand field.

For example, consider a 32-bit instruction set. If we have 64 instructions and 32 registers, this means we need 6 bits for the opcode (since $2^6 = 64$) and 5 bits for the register field (since $2^5 = 32$). We can have an opcode and three registers and consume $6 + 5 + 5 + 5 = 21$ bits. This leaves 11 more bits for other fields. Some instruction sets have *shift* fields which can allow limited-sized registers to represent larger numbers through shifting, that is, multiplying, them. The sky is really the limit here, and great human creativity goes into specializing all these bits. The key thing, though, is to realize everything is about tradeoffs and a balance must be struck between the size of operand fields and their effect on the overall architecture of the computing system.

Hex Code

Instructions are often represented as hexadecimal numbers because, as we discussed in Chap. 7, hex notation is used to compactly write and hold in our minds long strings of 1's and 0's. To write an instruction in hex code is to figure out what the opcode is for the given operation, the appropriate registers or immediate operand to encode, and string together all the 1's and 0's. Then, just break it up into groups of four bits and assign the hex values accordingly. In the case where the instruction is not a multiple of four bits in size, we start with the least significant bit and pad the left with zeroes as required.

Exercises

10.1. Suppose you have a 16-bit instruction encoding and have 32 general-purpose registers and use 8-bit numbers.

(a) If you have a 6-bit opcode, how many operand address fields can you have?
(b) If you have a 6-bit opcode, how many operand number fields can you have?
(c) How many opcodes can you have if you have an operand address field and an operand number field?
(d) How many three-address instructions can there be?

10.2. Suppose you have a 20-bit instruction encoding and have 16 general purpose registers and use 8-bit numbers.

(a) How many instructions can have two operand address fields and one operand number field?
(b) If you have an 8-bit opcode, how many operand address fields can you have?
(c) How many instructions can have two operand number fields?

10.3. Suppose the PC = x4520 at the beginning of the Fetch stage. What values are in the PC and IR registers at the beginning of the Decode stage?

10.4. Suppose the PC = 3A4BH and IR = x265C during the Decode stage. If the instruction encoding uses the first 8 bits for the opcode and the next 8 bits for two 4-bit operand addresses, (a) what registers are being used in this instruction and (b) at what memory address was this instruction located?

10.5. If IR = x8B2D and the instruction encoding uses the first four bits for the opcode, the next four bits for a register address, and the last 8 bits for an operand number, what register and number are being used in this instruction? Furthermore, if the opcode for the `ADD Rn, #N` instruction is 1000, then what is the actual instruction in the IR?

10.6. A computer with 32 general-purpose registers R0-R31 requires _____ bits for the operand register address field of the instruction encoding.

10.7. Suppose an instruction of the form **`MOV Rn, #N`** has the encoding x**9CA3**. If the opcode takes up the first 5 bits, the register address field the next 3 bits, and the operand number N the last 8 bits, then what is the complete instruction given by the hex encoding?

10.8. Suppose an instruction of the form **`MOV Rn, #N`** has the hex encoding x**46EA2**. If the opcode takes up the first 7 bits, the register address field the next 5 bits, and the operand number N the last 8 bits, then what is the complete instruction given by this hex encoding?

10.9. Fill in the blanks: During the Fetch stage of the instruction cycle, we first send the _____ of the instruction to be fetched to the memory unit. We then retrieve the instruction encoding and place it in the _____ . Finally, we _____ the _____ so that we are ready to Fetch the next instruction.

10.10. Consider the following instruction set:

```
MOV A, #N loads a value N into A
MOV Rn, A  copies the contents of A into Rn
MOV A, Rn  copies the contents of Rn into A
ADD A, Rn  adds A to Rn and stores the sum in A
```

Sequence these instructions (i.e. write a program) to accomplish the following tasks:

(a) Put the number 5 in R2 and add 7 to it, storing the result in A.
(b) Calculate the sum of 215 and 196, storing the result in R4.

10.11. Remember that, although we are designing computers in this class, a knowledge of programming is vital for the development of hardware. Since computers need to actually compute things from time to time, it is important to understand how instruction sets handle signed arithmetic. Use the instructions given below to write short programs involving two's complement. Note that the instruction set does not have a subtract instruction, so you must subtract by adding the two's complement. You will need to carefully apply your knowledge of computer arithmetic in order to get these instructions to do what you want them to do.

```
MOV A, #N loads a value N into A
MOV Rn, A  copies the contents of A into Rn
MOV A, Rn  copies the contents of Rn into A
ADD A, Rn  adds A to Rn and stores the sum in A
CPL A      calculates the one's complement of A
```

N must be a positive number; you cannot load negative numbers directly into A. You calculate the one's complement by inverting each bit of a number. For example, the one's complement of 1100101 is 0011010.

(a) Load the value −15 into register R2.
(b) Calculate the two's complement of the value in R5 (and store it back in R5.)
(c) Subtract the value in R5 from the value in R7 and store the result in R2.
(d) Subtract 10 from the two's complement of the value in R4 (and store the result in R4.)

While writing these programs, remember the easy-to-forget but absolutely fundamental fact that the computer doesn't "know" whether the number in a register is signed or unsigned.

Since two's complement assigns the large binary values to the negative numbers, what is really happening when we "subtract" is we are adding such a big value that the number "rolls over" and starts over again at 0.

So, as a programmer, we must understand all of this in order to effectively sequence instructions which use arithmetic. It is certainly worth your while to fully internalize exactly how two's = complement works and why it has become the standard.

10.12. For the following questions, assume we have 16 total bits for the instruction encoding and 16 general purpose registers (so the *operand address* field requires 4 bits).

(a) How many different opcodes are available for three-address instructions?
(b) If we require 10 bits for the opcode field, how many address fields can we have?
(c) If our numbers are 8 bits wide, then how many opcodes can we have if we have an instruction such as `MOV Rn, #N` which moves an actual number into a register? This instruction would require both an operand address field and a field for the operand number itself.

10.13. A computer has to bring the instruction from memory into the processor (Fetch), ready the operands (Decode), and then execute the instruction (Execute.) These stages—Fetch, Decode, Execute—make up the *instruction cycle*.

During Fetch, a register called the Program Counter (PC), which holds the address of the instruction being fetched, needs to be incremented so that it now holds the address of the instruction to be fetched next time. A register called the Instruction Register (IR) gets updated with the actual 1's and 0's encoding of the instruction.

(a) If, prior to the Fetch stage, PC = x2B49 and the instruction at location 2B49H has the encoding xA4C7, what are the values of the PC and the IR registers after the Fetch stage has completed?
(b) Suppose we have 16 general-purpose registers R0-R15 and the last 8 bits of the instruction contain address fields for them. In the previous example, what two registers is the instruction accessing?

(c) Suppose again we have 16 general-purpose registers but now the instruction encoding is as follows: first four bits for the opcode, next four bits for the register address, and the last 8 bits are for the actual number to be used. If the opcode 1010 corresponds to the `MOV` instruction which copies a number into a register, write the actual instruction being accessed in part (a) above.

10.14. Suppose we have a computer with 128 general purpose registers,16 M-words of memory, and 58 opcodes. How many bits does it take to encode each of the following? (Hint: $M = 2^{20}$.)

(a) an instruction with 2 register operands
(b) an instruction with 1 register operand and 1 direct operand

10.15. Fill in the blank: A computer with 64 general-purpose registers R0-R63 requires _____ bits for the operand register address field of the instruction encoding.

10.16. Suppose an instruction of the form `MOV Rn, #N` has the hex encoding x46EA2. If the opcode takes up the first 7 bits, the register address field the next 5 bits, and the operand number N the last 8 bits, then what is the complete instruction given by this hex encoding?

10.17. Give two operations that are performed by the computer during the Fetch cycle

10.18. Consider the following instruction set:

```
MOV A, #N   loads a value N into A
MOV B, A    copies the contents of A into B
MOV C, B    copies the contents of B into C
MOV A, C    copies the contents of C into A
ADD A, B    adds A to B and stores the sum in A
SUB C, B    subtracts B from C and stores the result in C
```

(a) Write a program which subtracts 7 from the value in C and stores the result in B.
(b) Write a program which adds the value in C to the value in A and stores the result back in C.

Notice how awkward this instruction set is to actually program with. You probably want to make some changes to the instructions that make the programming easier. Remember that the hardware (datapath) has to actually physically be able to execute each instruction. Software and hardware go hand in hand. In Chap. 15 we'll look at how to build datapaths to execute these instructions. So, keep this programming work in mind as you study how to specialize our digital logic components to accommodate the new instructions you would like to have available. Once you start along this path you are well on your way to designing your own computer!

10.19. Consider the following instruction set:

```
MOV A, #N Puts the value N into A
MOV B, A  Copies A into B
MOV Rn, A Copies A into Rn
MOV A, Rn Copies Rn into A
MOV B, Rn Copies Rn into B
ADD A, B  Adds A and B and stores the sum in A
```

Note that the n in Rn is an index that should be replaced with a number to access one of the registers R0 through R7.
Write programs using instructions from this set to accomplish the following tasks:

(a) Store the value 17 in B and 122 in R5.
(b) Add the value in R2 to the value in R7 and store the result in R4.
(c) Add the three values 12, 25, and 101 together and store the result in R7. Try to do it as efficiently as possible.
(d) Add 10 to the value stored in R1.

10.20. Consider the following instruction set:

```
MOV Ra, #N    Puts the value N in Ra
MOV Ra, Rb    Copies Rb into Ra
ADD Ra, Rb, Rc Adds Ra + Rb + Rc and puts the sum in Ra
ADD Ra, #N    Adds N to Ra and stores the sum in Ra
CLR Ra        Sets the value of Ra to 0.
```

Note that the a, b, and c in Ra, Rb, Rc are indices that should be replaced with a number to access one of the registers R0 through R15.
Write programs using instructions from this set to accomplish the following tasks.

(a) Store the value 17 in R4 and the value 122 in R14.
(b) Add the value in R2 to the value in R7 and store the result in R4.
(c) Add the three values 12, 25, and 101 together and store the result in R7. Try to do it as efficiently as possible.
(d) Add 10 the value stored in R1.

Chapter 11
Sequential Logic Elements

When we write our programs, we take for granted the fact that we can declare a variable x and assign it some value and be assured that it will retain said value until we specify it is to be modified.

This sets us a digital design task: how do we connect our logic gates in such a way that the resulting circuit will **store** a value?

The SR Latch

Thus far we've seen that our circuits have three basic parts: inputs, logic gates, and outputs. The flow of signals is quite clear: start with the input, flow through the logic to implement a particular computation, and then output the result. If we want a circuit to *have memory* this means we want the result to be retained. The way to do this is take the output and pump it back into the system: build a circuit where the output wires are also the input wires. Consider the example in Fig. 11.1.

This circuit is called an **SR Latch** and is capable of storing one bit worth of information. Let's see how it works. First, suppose we have S = 1 and R = 0. We don't have to care what Q and $\bar{Q}$ are before we apply these inputs, because inputting S = 1 to the NOR gate will force the $\bar{Q}$ output to be 0 regardless of what Q was. With $\bar{Q} = 0$ and R = 0 we see that the top NOR gate results in $Q = 1$. Figure 11.2 illustrates this behavior.

If we apply R = 1 and S = 0, the symmetry of the circuit ensures we get the opposite result: $Q = 0$ and $\bar{Q} = 1$.

What happens if we apply R = 0 and S = 0? Now we *do* have to care what the value of Q and $\bar{Q}$ are ahead of time. If $Q = 1$then the output of the lower NOR gate becomes 0 which forces the output of the top NOR gate to 1 which maintains the value $Q = 1$. Again, the symmetry of the latch tells us that if $\bar{Q} = 1$ then it will also retain its value. So, we find that an input of R = 0 and S = 0 maintains the current value of Q stored in the circuit.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_11

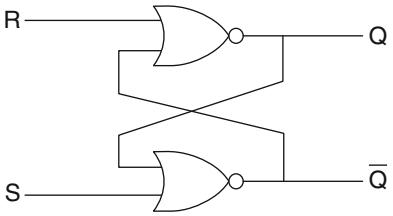


Fig. 11.1 SR Latch

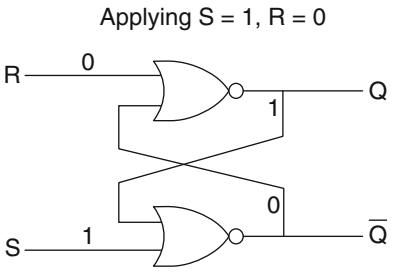


Fig. 11.2 Operation of SR Latch

Table 11.1 Truth table for the SR latch.

S	R	Q	
0	**0**	**Q**	**hold**
0	**1**	**0**	**reset**
1	**0**	**1**	**set**
1	**1**	**--**	

The S and the R stand for **set** and **reset**, respectively, and we call the act of storing a 1 in a latch *setting* it and the act of storing a 0 in a latch *resetting* or *clearing* it. If we don't change the value we say we are *holding* the value.

What if we try to set and reset at the same time? Is that awesome or dumb? We can see from the diagram that sending two 1's gives us two 0's out of the NOR gates which means both Q and $\bar{Q}$ are 0. So, I guess it's dumb. What's worse, though, is that if we then go from S = R = 1 to S = R = 0 we have no idea what's going to happen! We end up with two 0's hitting one of the NOR gates, which turns its output to 1 which leads to the other being eternally 0. But. . .which output, Q or $\bar{Q}$, is set to 1 and which has to remain 0? We actually don't know! It depends on the details of the physical implementation! We can draw the circuit with perfect distances between the inputs and NOR gates but in the real implementation there will be some engineering imperfection that means one of the signals will propagate just *this much* faster than the other and will end up dominating. This situation is called a **race condition** and is something we want to avoid. So, we'll agree not to try to set and reset the SR Latch at the same time.

The truth table for the SR Latch, then, is as shown in Table 11.1.

The circuit diagram we'll use to represent the SR Latch is that of Fig. 11.3.

The logic we've been working with up to this point is called **combinational logic** because we are certain that the output of the gates is the response to a combination of

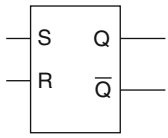


Fig. 11.3 SR latch block diagram

the current inputs. The inputs that were sent to the device last Tuesday do not matter. All we care about are the current combination of inputs. In contrast, logic circuits with feedback like the SR Latch are classed as **sequential logic** because the output depends not only on the current inputs to the system, but also on the sequence of past inputs. When we send S = 0 and R = 0 as input to our SR Latch, we have *no idea* what the output will be! The output is Q! What is Q? Well, Q depends on whether the past inputs were set or reset. To determine the SR Latch's output we have to know the sequence of inputs leading up to the hold signal. Other more complex sequential logic circuits can depend on much longer sequences.

Timing

We call the value stored in the latch the **state** of the system and we use the term **state machine** to refer to circuits we build based on tracking state. This is of fundamental importance because the entire intellectual basis for how we approach imperative programming is tied up in this concept of *state*. The values in all the variables in our program, and in memory, and the PC and IR values, and status register values, and a host of other things all combine to make up the computer's *state* at any given moment. In order for our program to run correctly, in order to execute statement after statement or instruction after instruction, we have to change the state in an orderly fashion. It does us no good if variable x is being updated when we're trying to assign a value to variable y. That would make debugging a nightmare. We just could not operate under those conditions. So, at the heart of what we're trying to accomplish here, building a computer, is the central problem of controlling when the *state* of our system updates.

The way we're going to solve this problem in our development here is through the imposition of a **synchronization signal**. This signal is going to be the supreme overlord of the entire computer system and no latch is going to be able to update its state unless the synchronization signal permits it. In this way, we are able to tame the chaos of analog circuitry and race conditions and create an environment in which we, the logical mavens that we are as we impeccably order the beauty of our programs, can thrive. After all, we use more semi-colons than the literature majors and can distinguish modus tollens from hypothetical syllogism better than the philosophers. We are not going to be held hostage to the whims of quarks and electrons (do they even exist? I think not!) and anything so crude as *wires* and *metal*. No, I assure you, our will to have our programs execute in a formal, rigorous, order shall not be thwarted. I introduce you to our friend, our ally, our trumpeter, the **clock.**

It's a curious thing, quartz. Look at it, and it's kind of pretty. Hit it and it hurts a bit. But if you hook it up to a battery and send electric current through it the quartz will vibrate at a controlled and predictable frequency. The device physicists call this an **oscillator** and we use these—made of quartz and other crystals—to power our synchronization, or clock, signals in our computers. The reliability of the oscillation of these materials permits us to move forward with our designs confident that our state updates will happen in concert.

In Fig. 11.4 we see a visual depiction of the output of a quartz oscillator.

We can see the value alternating between logic-1 and logic-0 at equal intervals. Each complete 1 and 0, or high and low, interval is called a **period** or a **clock cycle.** The **frequency** of a clock tells us how many periods are in a second. Since we will tie state updates inside our computer to a single clock cycle, this means a higher frequency clock will enable more state updates per second which means we can perform more computations per second.

But when, exactly, are we going to let our states update? We have four choices, as there are four distinct regions within a clock cycle, as shown in Fig. 11.5.

Most of the time the signal is either at a logic-1 (high) or logic-0 (low) level. But for a near infinitesimal moment the signal is in transition. We call the transition from high to low the **falling edge** and the transition from low to high the **rising edge**. It turns out that we can build our latches such that they update in any of the four regions we may want. The schematic for these devices is slightly different to show the type of timing it's keyed for. See Fig. 11.6.

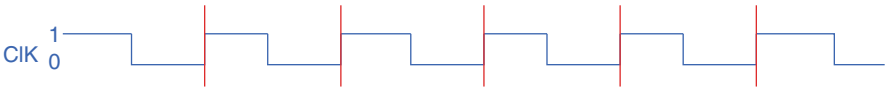


Fig. 11.4 Square wave

Fig. 11.5 Anatomy of a clock cycle

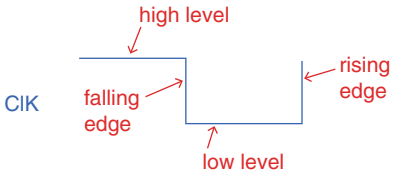
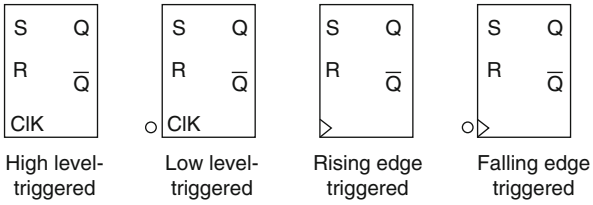


Fig. 11.6 Timing methodologies



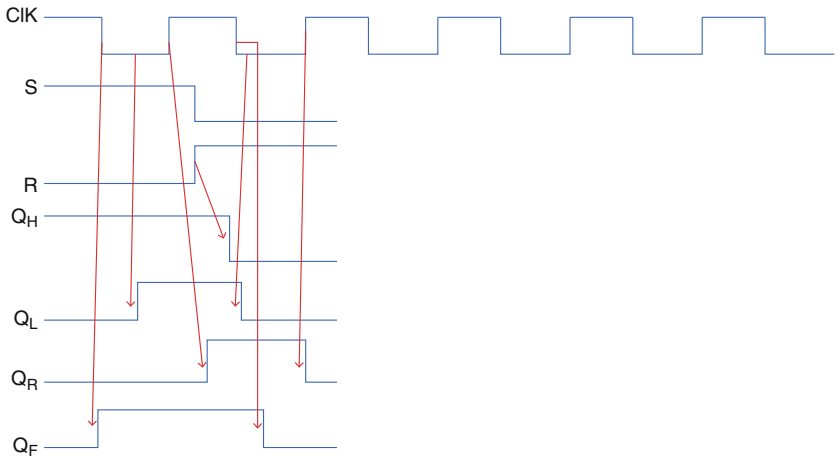


Fig. 11.7 Timing diagram

We can visualize the timing behavior of these latches in what we call a **timing diagram**. Suppose Q_H is the output of the high level-triggered device, $Q_L$is the output of the low level-triggered device, Q_R is the output of the rising edge triggered device, and Q_F is the output of the falling edge triggered device. Assume all the outputs start out 0. Then we can see how each output responds to changes in the clock signal in Fig. 11.7.

We can see that while, eventually, all these devices respond to the set and reset signals in the way we would expect, the exact timing of each of them is different. The high level triggered device updates first because it is checking for S at the very beginning of the clock cycle before any of the other devices care at all. Next to change is the falling edge triggered device because it checks the S and R signals when the clock transitions from 1 to 0. Then the low-leveled trigger device looks to see what is up with S and R, and finally the rising edge triggered device catches up with the others at the very last moment available in the clock cycle.

It's important to understand what is going on with the diagram. Look it over until you can tell this story and explain the second clock cycle and the transition of all the devices back to their starting value of 0.

So, with all these choices, which is best for us? It turns out that the level-triggered devices are not great for what we're trying to do. Since they expose the state to change for half the clock cycle, if we use them we end up with values updating multiple times a cycle, and that can lead to undesirable behavior. We are much more concerned with the edge-triggered devices. They are nearly universally used to implement clock signals in digital systems. Which you choose—rising or falling edge—depends on factors contingent on other things going on in the system beyond the immediate design area. For the rest of this text, we're going to default to the rising edge triggered device, but not much changes if we use the falling edge triggered device instead.

Building a Register

Putting this all together, let's finally build a **register**! We want to be able to store large 32- and 64-bit numbers, not just a single boring bit. To do this, we need to network together a number of one-bit storage latches. Let's first do it with our SR Latch.

We want the register to be able to update a value, but we don't want it updating automatically every single clock cycle. We want to use our instructions to control exactly when the value updates. What we need is a **control signal** to help out here. We call it a **load** signal and denote it **LD**. When LD = 1, we want to allow the device to update. When LD = 0, we want it to hold its value. So, we can build a bit of a truth table to figure out the logic necessary to make this happen. See Table 11.2.

We want to *set* when X = 1 and LD = 1, so we can write the logic function $S = X \cdot LD$ to express that behavior. We want to *reset* when X = 0 and LD = 1, so we can write the logic function $R = \bar{X} \cdot LD$ to express that behavior.

So, the design of a 4-bit register using SR Latches to store the state looks like Fig. 11.8:

To make a larger register we just add more SR Latches.

We don't have to draw the register using the latches every time we need to put one into a design. The schematic of Fig. 11.9 will suffice.

Table 11.2 Excitation table for the SR latch

X	LD	S	R
0	**0**	0	0
0	**1**	0	1
1	**0**	0	0
1	**1**	1	0

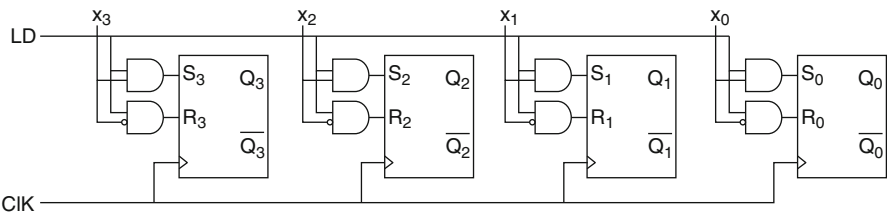
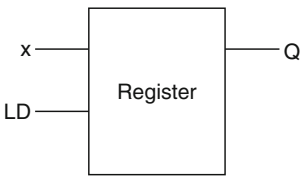


Fig. 11.8 Register design with SR latches

Fig. 11.9 Block diagram of the register



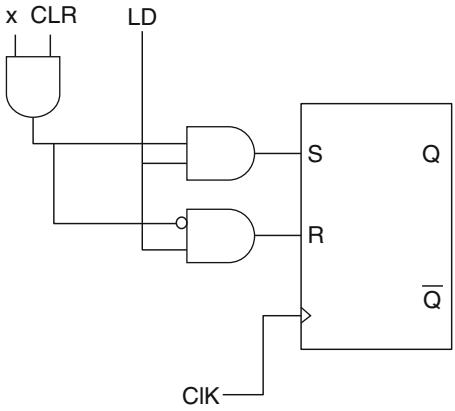


Fig. 11.10 Bit slice of CLR operation

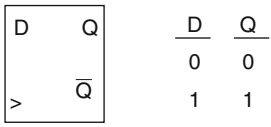


Fig. 11.11 D flip flop block diagram and truth table

We'll often have a lot of registers in our system, and we give them different names such as R_0, R_1, R_2, etc., or letters like A or B, or special register such as PC and IR. In these cases we'll label the specific load signal for each register with an underscore and the name: LD_A or LD_PC, say.

We can add features to registers as well. One common thing we often want to do is quickly clear the values in a register. We can add a new control signal CLR to accomplish this. We want CLR to force a reset regardless of what X is. One way to accomplish this is to use an AND gate so that a one-bit slice of the register would appear as in Fig. 11.10.

Now whenever CLR = 0 the register will update with all 0's regardless of what X happens to be. The LD still has to be 1 to clear, but that functionality could be changed as well.

Other Flip Flops: D, JK, and T

Keeping both S and R in mind complicates the design of our state machines. While ultimately SR Latches have the potential to lead to better logic implementations, typically modern machines are designed entirely using what is called the D Flip Flop, or DFF. It's a simpler design, as seen in Fig. 11.11.

The DFF simply stores whatever is the input. You don't have to go through a bunch of logic deciding what gates to employ for S and R. How do we build a register with LD input with DFF's if they don't have the ability to hold? The answer to that lies in the next chapter, as we'll need to use a multiplexer to accomplish that.

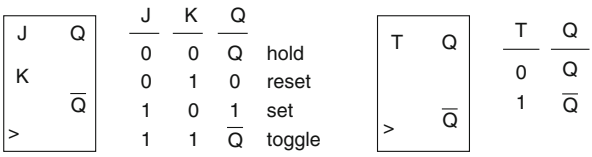


Fig. 11.12 JK and T flip flops

Table 11.3 State table

Present state	Inputs		Next state
Q	J	K	D
0	0	0	0
	0	1	0
	1	0	1
	1	1	1
1	0	0	1
	0	1	0
	1	0	1
	1	1	0

Before we go, let's introduce some more members of the flip-flop family: the JK and the T. Their block diagrams and truth tables seen in Fig. 11.12.

You can see these are based on a new function: toggling. To toggle a bit means to "flip" it or complement it: 1 toggled becomes 0 and 0 toggled becomes 1. The JK solves the problem the SR had by making the 11 inputs toggle the state rather than crashing the system; otherwise, it's identical in function to the SR. The T is like the D in that it only has one input but the input toggles the state rather than directly setting or resetting it.

We'll see the uses for these flip flops as we continue to use these devices to design more complex state machines.

The State Machine Design Process

One thing we'll look at here is that you can build any of the flip flops from any of the others. For example, here's how we can use DFF's to construct a JK Flip flop.

The first step is to build what is called a **state table** to organize all our thoughts on the matter. Since these devices store a *state*, we need to specify to what state the machine transitions upon application of every combination of inputs to every possible state. We need three columns present state, inputs, and next state to track this. In our case we have a 1-bit state and a 2-bit input (since we want to build the JK flip flop.) This is put together in Table 11.3.

The present state is represented by Q as that is the output of the DFF. The next state is represented by D because that is what we need to input to the DFF to affect the state transition we want in our design. The inputs J and K indicate the functionality we want to see in our device: when they are both 0 the next state is

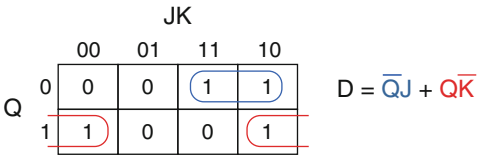


Fig. 11.13 K-map for D from Table 11.3

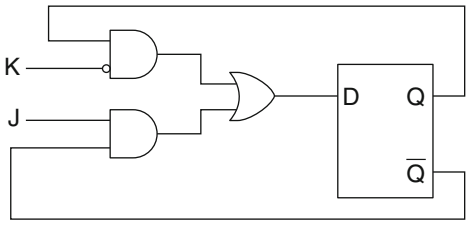


Fig. 11.14 Circuit implementation of state machine specified in Table 11.3

A	B	Function
0	0	hold
0	1	toggle
1	0	reset
1	1	set

Fig. 11.15 AB flip flop specifications

the same as the present state, when J = 1 we set so the next state is always 1, when K = 0 we reset so the next state is always 0. Finally, when both J and K are 1 we toggle the state, changing from 0 to 1 and from 1 to 0. Make sure you understand how this table describes the system we're building. We're going to use this exact same framework throughout the entire text to design all of our state machines. Any energy you put into understanding it now at the most basic level will pay off.

After we specify our function, we now need to synthesize equations for D. See Fig. 11.13.

Figure 11.14 shows the circuit implementation.

In principle, we can design any sort of flip flop out of any other. There is one issue to keep in mind when working with memory elements that are not DFF's, though: for these, the function we're specifying can't be just the next state. To transition from 0 to 1 with an SR flip flop requires S = 1 and R = 0, for example, and this is slightly more complex (it's still not that hard, really) than our table above illustrating a design based on DFF's. (It is partially for this reason that the DFF is the dominant flip flop used.)

Let's work an example. Let's build the AB flip flop device seen in Fig. 11.15 from an SR latch.

The state table is seen in Table 11.4.

You should check the table entries against the AB latch's control table to make sure you understand how it was constructed. Notice we no longer label the Next State as D. This is because we're not using a D latch and will have to think a bit harder about what output functions we need to specify in our device.

Table 11.4 State table for building the AB flip flop of Fig. 11.15 out of an SR latch

Present state	Inputs		Next state
Q	A	B	
0	0	0	0
	0	1	1
	1	0	0
	1	1	1
1	0	0	0
	0	1	0
	1	0	0
	1	1	1

Table 11.5 SR Implementation table

Q_t	Q_{t+1}	S	R
0	0	0	d
0	1	1	0
1	0	0	1
1	1	d	0

Table 11.6 State table

Present state	Inputs		Next state	Implementation	
Q	A	B		S	R
0	0	0	0	0	d
	0	1	1	1	0
	1	0	0	0	d
	1	1	1	1	0
1	0	0	0	0	1
	0	1	0	0	1
	1	0	0	0	1
	1	1	1	d	0

For each state transition we may need we need to ask ourselves "what are the required values of S and R to make that transition happen?" For example, if we're transitioning from state 0 to state 1 we need to set the bit and issue $S = 1$ and $R = 0$. However, if we're transitioning from state 0 to state 0, we have to choices: to reset or to hold. So we could have $S = 0$ and $R = 0$ or we could have $S = 0$ and $R = 1$. Really, as long as $S = 0$ we *don't care* what R is. What we are building is called the **implementation table** (or sometimes the activation table) for the SR flip flop. It tells us what S and R should be for whatever transition we need. It can be seen in Table 11.5.

Go through the logic of this table to be sure you understand it. We use Q_t to represent the present state (state at time t) and Q_{t+1} to represent the next state we want (state at time t + 1).

With this, we are ready to specify the **implementation functions** for our machine. We can update the state table (Table 11.6.)

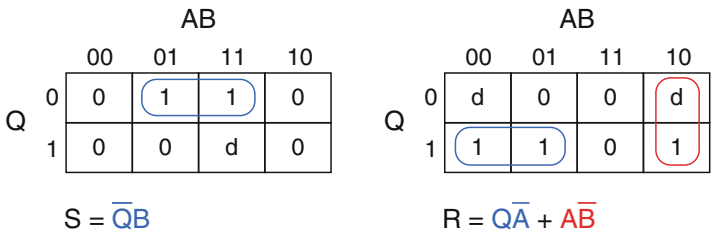


Fig. 11.16 K-maps for the state machine of Table 11.6

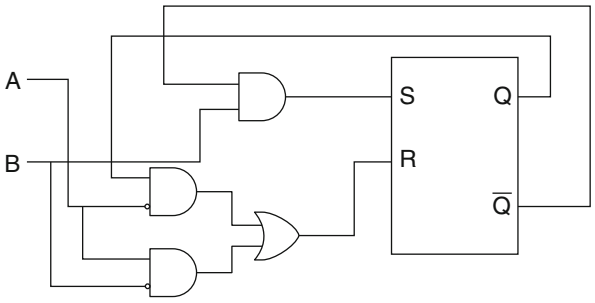


Fig. 11.17 Circuit diagram for state machine specified in Table 11.6

Table 11.7 Implementation tables for D, SR, JK, and T flip flops

Q_t	Q_{t+1}	D	S	R	J	K	T
0	**0**	0	0	d	0	d	0
0	**1**	1	1	0	1	d	1
1	**0**	0	0	1	d	1	1
1	**1**	1	d	0	d	0	0

We fill in the implementations by comparing the present state value to the next state value for each row (we ignore inputs for this purpose.) Since the first row requires a transition from present state 0 to next state 0 we need S = 0 and R = d (we don't care what R is) in the table. Check to make sure you can see how this table was built from the SR implementation table.

Now it's all but over, as we are next tasked with synthesizing our output functions S and R. Figure 11.16 shows the relevant K-maps.

The full circuit diagram is seen in Fig. 11.17.

We'll need to reference the implementation tables for the various flip flops to assist this process. For completeness, Table 11.7 collects all four of them:

Notice there are more don't cares in the JK implementation? This is due to the fact that we can now toggle as well as reset to transition from 0 to 0 and we can toggle as well as set to transition from 1 to 1. The power of JK is that it permits simpler logic in our next state equations.

For example, here's the state table for our above AB device using JK flip flops instead of SR:

The synthesized equations are now seen in Fig. 11.18.

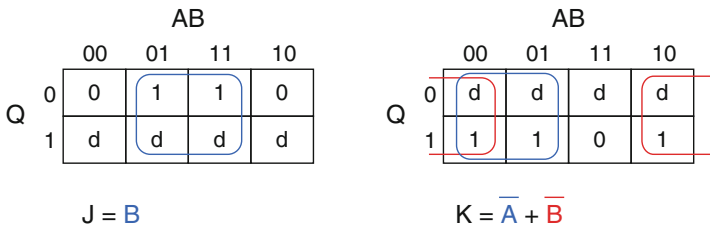


Fig. 11.18 K-maps for J and K output specified in Table 11.8

Table 11.8 The AB device of Fig. 11.15 implemented with JK.

Present state	Inputs		Next state	Implementation	
Q	A	B		J	K
0	0	0	0	0	d
	0	1	1	1	d
	1	0	0	0	d
	1	1	1	1	d
1	0	0	0	d	1
	0	1	0	d	1
	1	0	0	d	1
	1	1	1	d	0

You can see we realize quite a savings from using these equations instead of the ones derived for the SR.

This process is the core thing we do when we build state machines, and state machine design is going to be a very important topic moving forward. It's recommended that you get comfortable with these very simple designs and the design process outlined here. If you do, you'll be set when we start having systems with more than two states and the inputs start representing quantities crucial to an algorithm or the running of an entire computer system.

Exercises

11.1 Produce the timing diagram for Q and $\bar{Q}$ for a level-triggered SR latch given the input signals in Fig. 11.19. Assume $Q = 0$ and $\bar{Q} = 1$ to start.

11.2 Produce the timing diagram for Q and $\bar{Q}$ for a pulse-triggered D flip flop given the input signals in Fig. 11.20. Assume $Q = 0$ and $\bar{Q} = 1$ to start.

11.3 Produce the timing diagram for Q and $\bar{Q}$ for a positive edge-triggered SR flip flop given the input signals in Fig. 11.21. Assume $Q = 0$ and $\bar{Q} = 1$ to start.

11.4 Produce the timing diagram for Q and $\bar{Q}$ for a negative edge-triggered JK flip flop given the input signals in Fig. 11.22. Assume $Q = 0$and $\bar{Q} = 1$ to start.

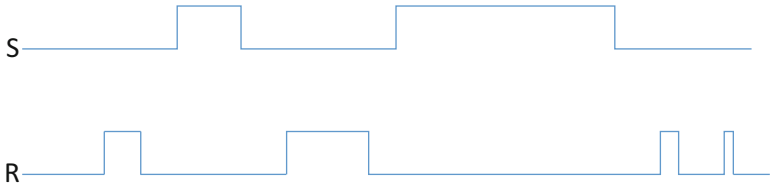


Fig. 11.19 Timing diagram for exercise 11.1

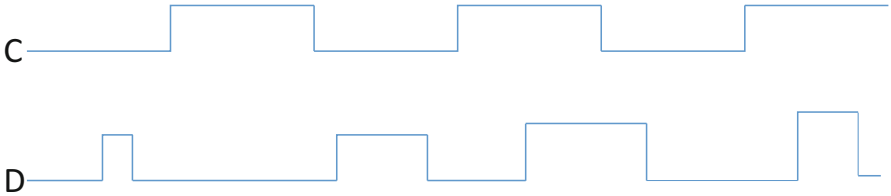


Fig. 11.20 Timing Diagram for Exercise 11.2

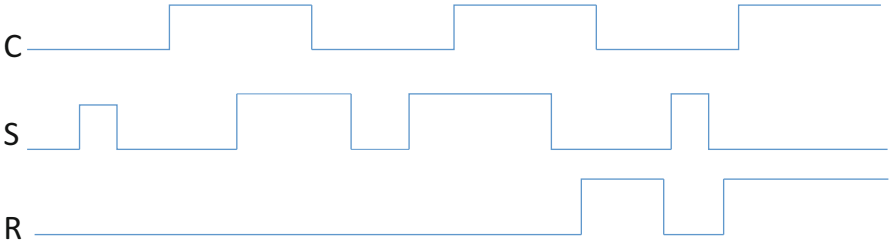


Fig. 11.21 Timing diagram for exercise 11.3

11.5 Draw the gate-level diagram for a DFF without enable. Use NAND gates.
11.6 Draw the gate-level diagram for a DFF without enable. Use NOR gates.
11.7 Draw the gate-level diagram for a DFF with enable. Use NAND gates.
11.8 Use level-triggered DFF's to build a positive-edge triggered DFF.
11.9 Use level-triggered DFF's to build a negative-edge triggered DFF.
11.10 Level-triggering makes sense—it's just like the enable signals on the decoders we worked with earlier in the course. Edge-triggering is different, however, as it forces the update to occur in a very specific point in time. Yet, edge-triggering is very powerful. Why do we use edge-triggered instead of level-triggering in most of our designs?

11.11 Build an S-R flip flop using only a toggle flip flop and combinational logic.
11.12 Complete the timing diagram of Fig. 11.23 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value.
11.13 Complete the timing diagram of Fig. 11.24 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value Fig. 11.24.

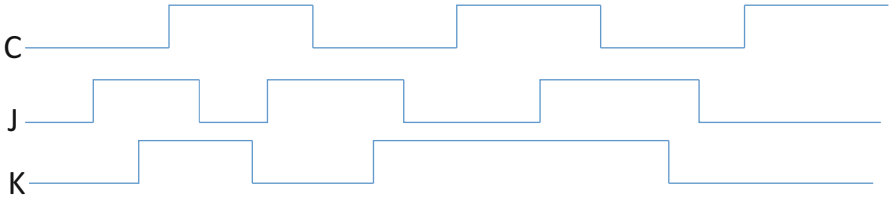


Fig. 11.22 Timing diagram for exercise 11.4

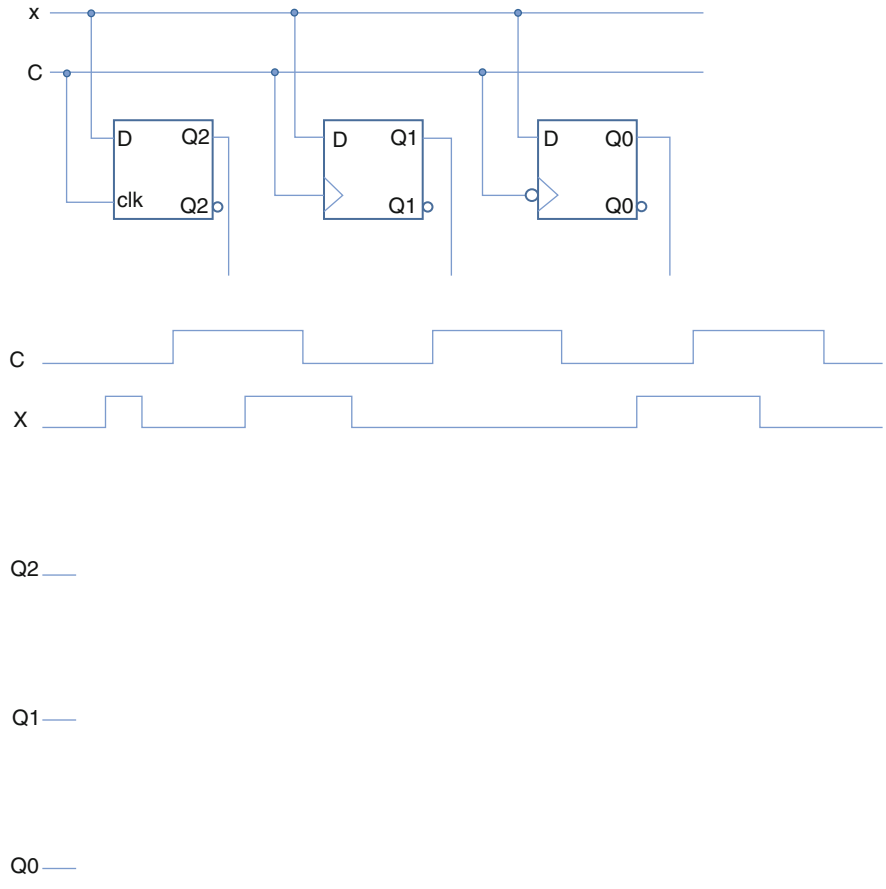


Fig. 11.23 Timing diagram for exercise 11.12

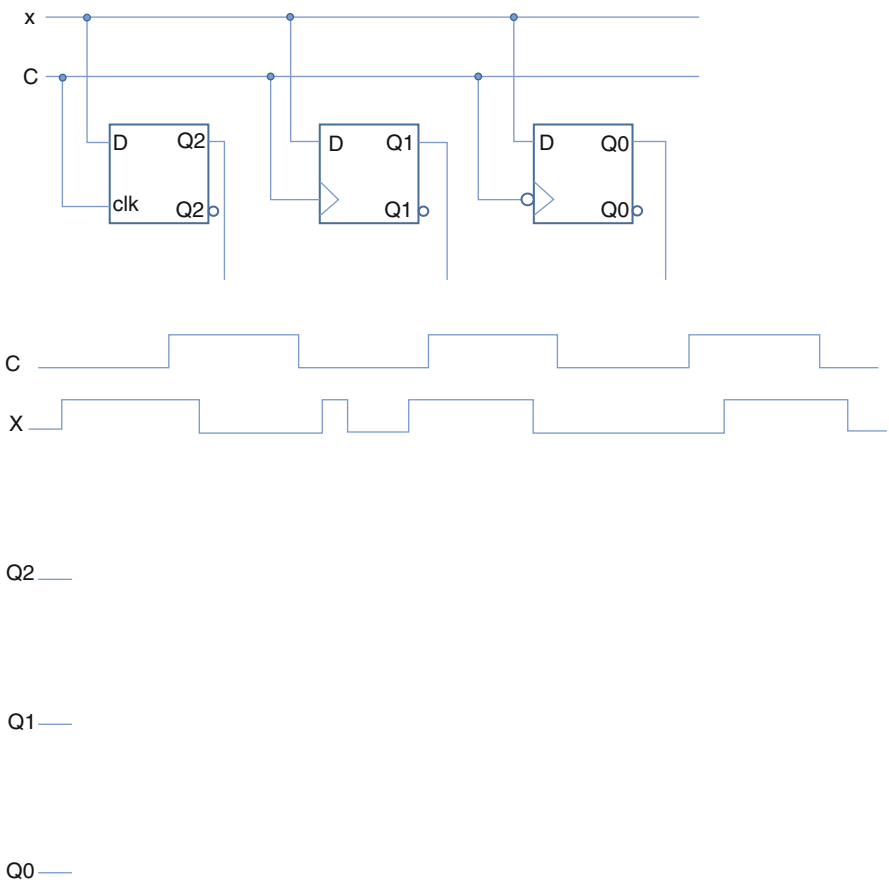


Fig. 11.24 Timing diagram for exercise 11.13

11.14 Complete the timing diagram of Fig. 11.25 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value.
11.15 Complete the timing diagram of Fig. 11.26 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value.
11.16 Complete the timing diagram of Fig. 11.27 for the given memory elements. Assume Q3, Q2, Q1, and Q0 all begin with a logic-0 value.
11.17 Complete the timing diagram of Fig. 11.28 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value.
11.18 Complete the timing diagram of Fig. 11.29 for the given memory elements. Assume Q2, Q1, and Q0 all begin with a logic-0 value. For the AB device, use the following description: 00 = Set, 01 = Toggle, 10 = Reset, and 11 = Hold.
11.19 Consider an AB flip flop which functions as illustrated in Table 11.9.

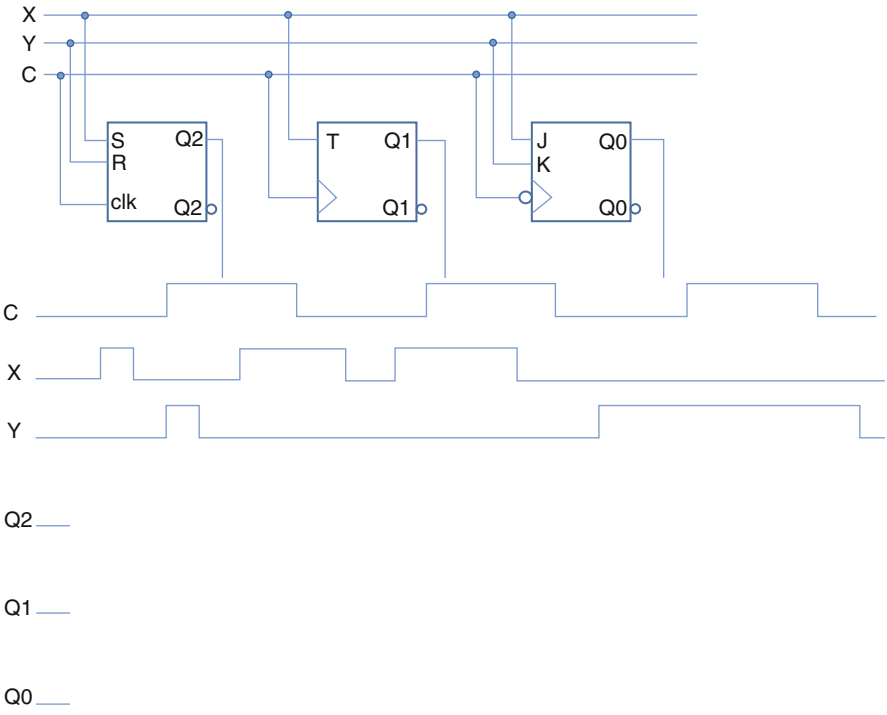


Fig. 11.25 Timing diagram for exercise 11.14

Build an AB device out of a T flip flop. Follow the same process you used in Problem 2 above but now your inputs are A and B and you need to solve for T to implement. Draw the resulting circuit.

11.20 If you load a register with a new value during a clock cycle, why can't you read this new value during the same cycle?

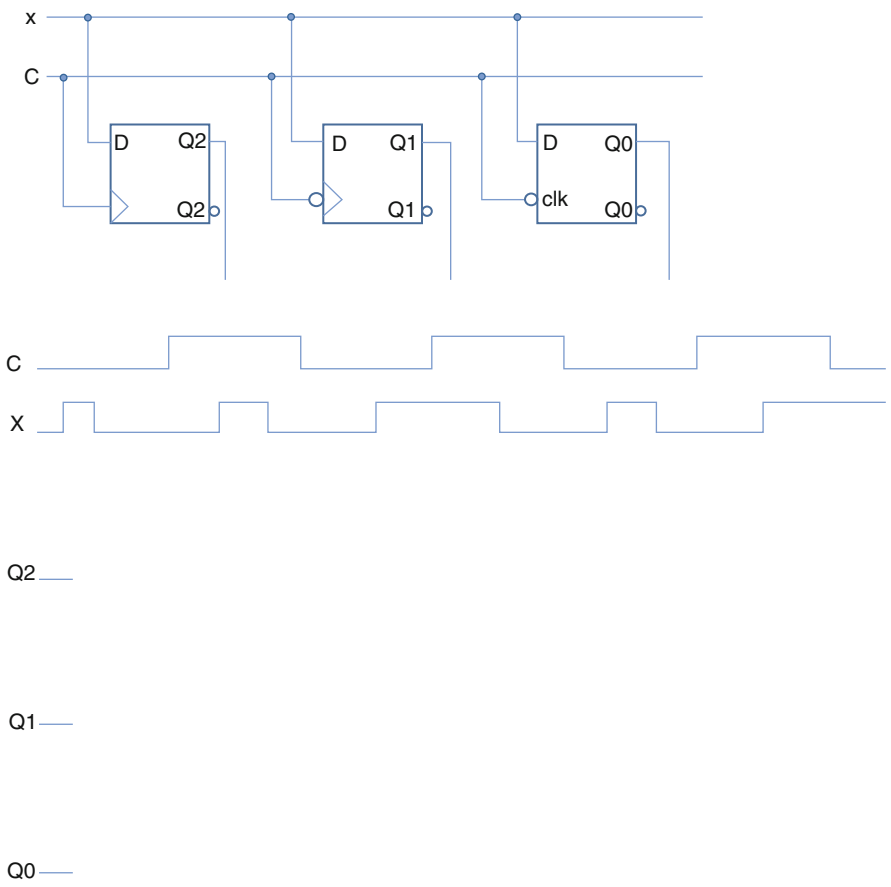


Fig. 11.26 Timing diagram for exercise 11.15

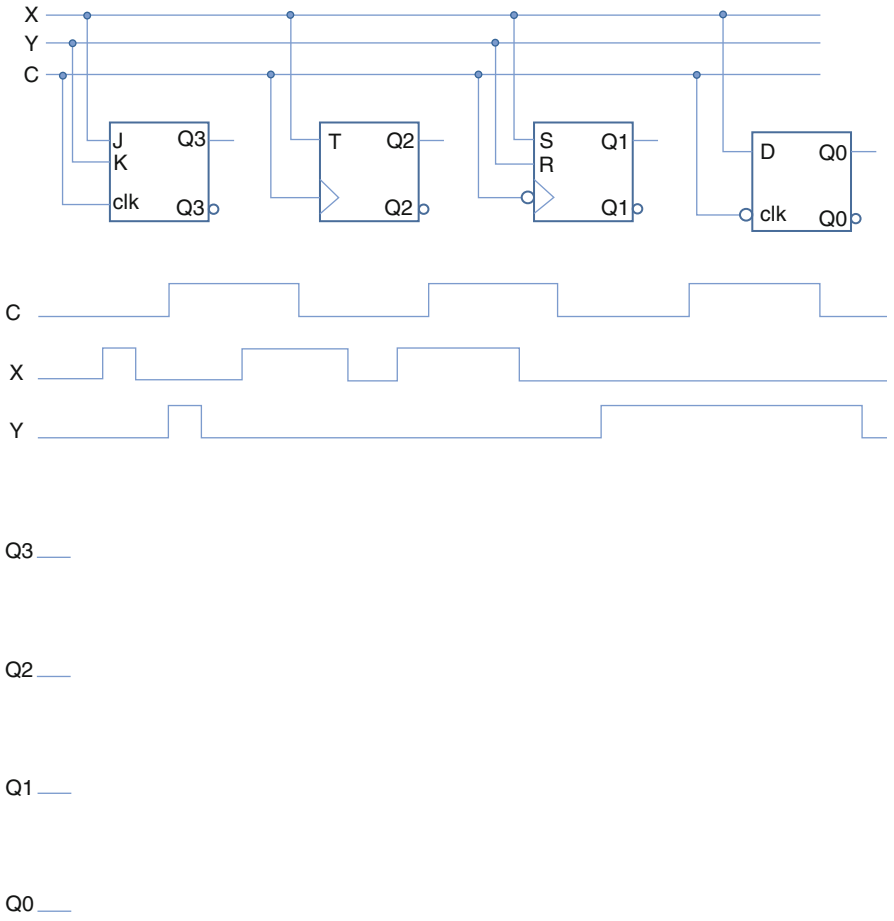


Fig. 11.27 Timing diagram for exercise 11.16

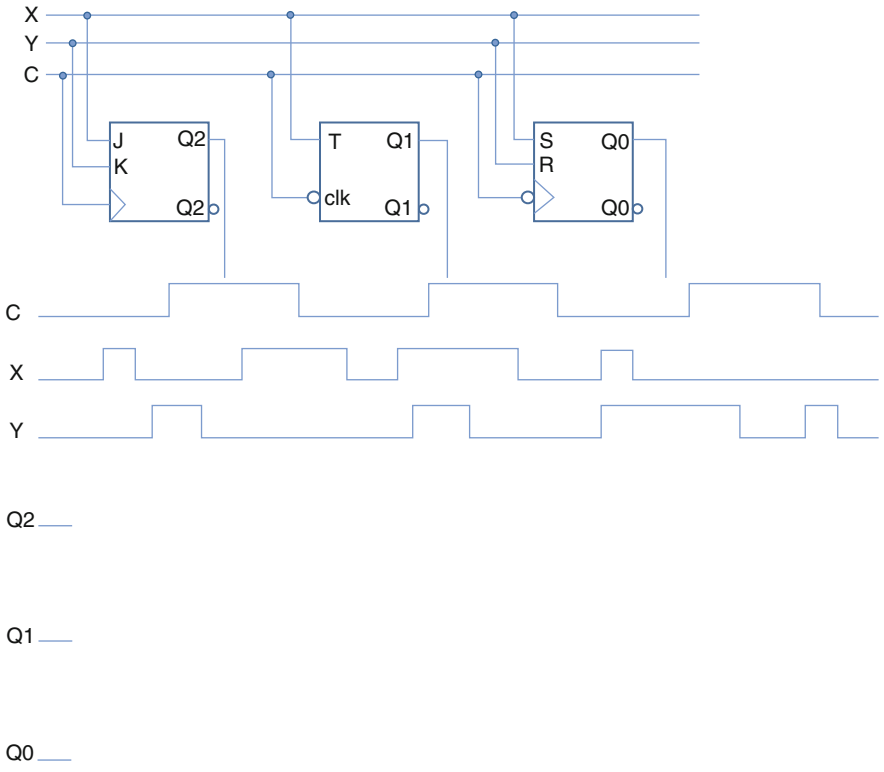


Fig. 11.28 Timing diagram for exercise 11.17

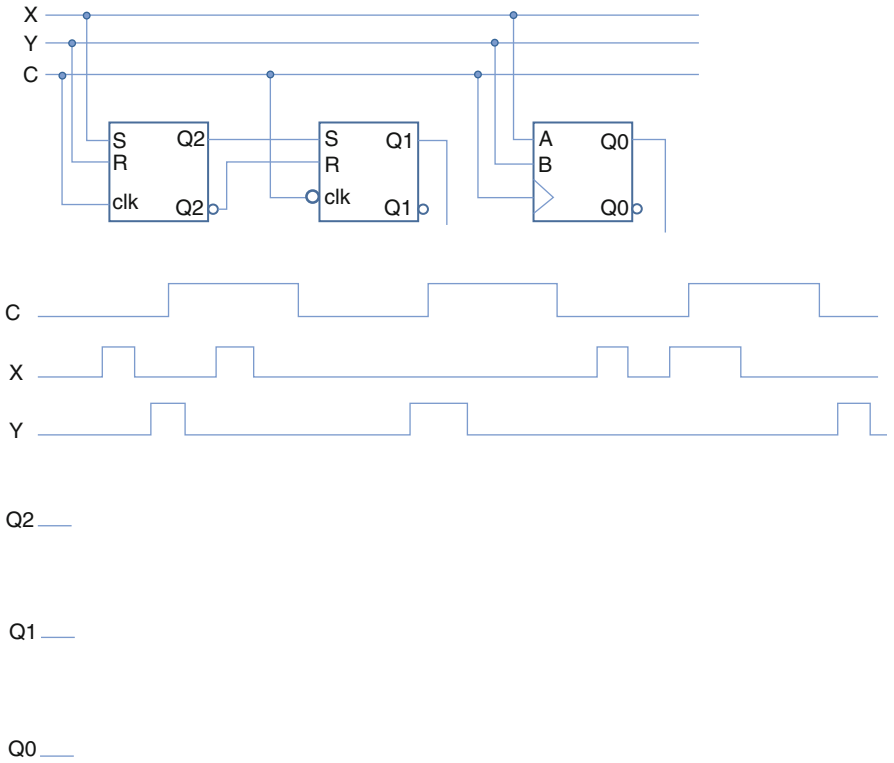


Fig. 11.29 Timing diagram for exercise 11.18

Table 11.9 AB flip flop for exercise 11.19

A	B	Function
0	0	set
0	1	hold
1	0	toggle
1	1	reset

Chapter 12
Multiplexers and Comparators

Now that we have a register and an adder, we can start building **datapaths** to implement our instructions. Suppose we have the instruction `ADD A, R1` which will compute A + R1 and store the result back in A. The circuit in Fig. 12.1 does this for us.

We can see from the diagram that the physical wires represented by the connections will transmit the value stored in both registers through the Adder and back into A. We want all of this to happen within the space of a single clock cycle so that the value in A won't change during the operation (awkward for the adder) and the final result stored back in A will be the correct sum and not an intermediate value that's not quite there. Throughout, we are going to assume our devices are designed compatibly in this regard. The details of this level of work are beyond the scope of what we're trying to accomplish here.

This circuit cannot implement the instruction `MOV A, R1` which will copy the value in R1 and move it into A. One approach to doing that would be to insert a new connection between R1's output and A's input, as seen in Fig. 12.2.

Danger! We are already using the input to A and we can't have multiple inputs to a register (well, I suppose technically we could but no one builds them that way because it would be way too inefficient and cumbersome.) We need a solution to this because it's going to be quite common for us to want to have multiple wires potentially able to serve as the input to some other device.

The solution to our problem is a **multiplexer.**

A multiplexer, or MUX, is a device that selects one of a number of inputs to pass through. Figure 12.3 shows block diagrams for devices of this class. The number of inputs can be whatever is needed in an application. An 8:1 MUX (pronounced "eight-to-one MUX") selects one among 8, a 4:1 MUX one among 4, etc. In addition to inputs, the MUX also accepts **control signals** that determine which of the inputs is passed through to the output. A MUX needs enough control signals to count all its inputs: an 8:1 MUX needs 3, a 4:1 MUX only needs 2, while a 32:1 MUX would require 5. That's it! It's a pretty simple, yet vitally effective,

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_12

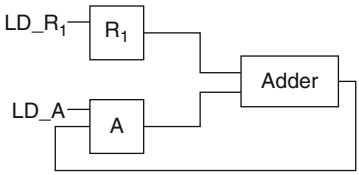


Fig. 12.1 Datapath for A = A + R1

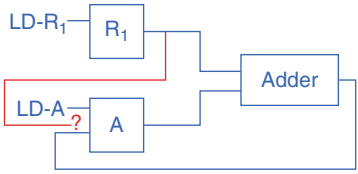


Fig. 12.2 Attempted datapath for A = R1 and A = A + R1

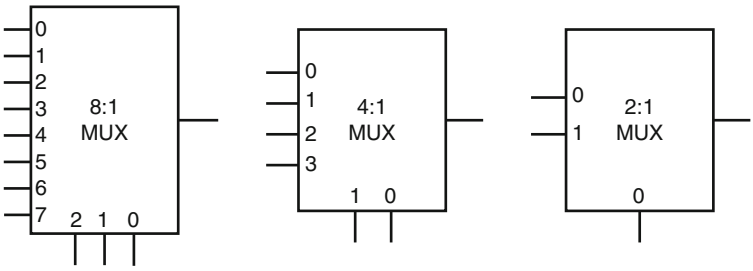


Fig. 12.3 Multiplexer block diagrams

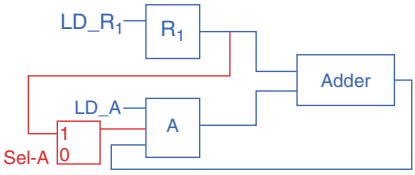


Fig. 12.4 Multiplexing the input to register A

component for our systems. We'll use one in just about everything we build from here on out.

In Fig. 12.4 we see our circuit updated with a MUX so it can implement both instructions `ADD A, R1` and `MOV A, R1`:

We need a new control signal Sel_A to work the MUX, but we're good to go now. We can select the ADD A, R1 instruction if we set Sel_A to 0 and we can choose the MOV A, R1 instruction if we set Sel_A to 1 (LD_A would also have to be 1 in both cases to permit updating of A.) This is crucial for you to understand, as it's our first step towards a huge world of computer datapath design.

Let's look inside the MUX to see the gates.

From Fig. 12.5 we can tell that it's pretty simple at this level: the inputs are tied to the select lines through AND gates which will select exactly one of them to be let through. More sophisticated encoding techniques can be applied. A strategy known

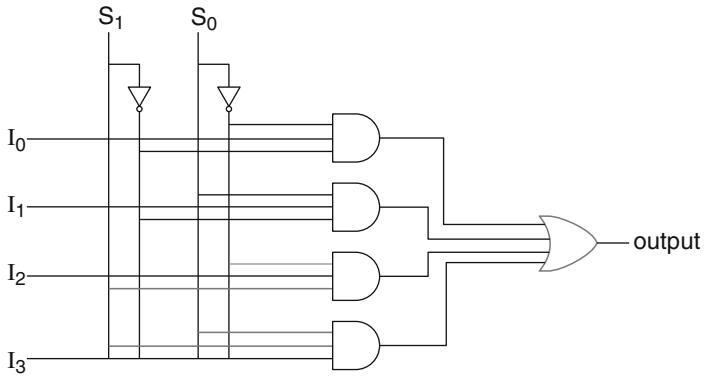
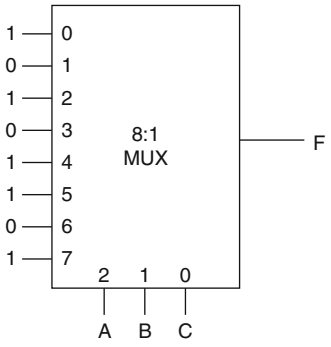


Fig. 12.5 Multiplexer gate-level diagram

Fig. 12.6 Implementing $F = \sum m(03, 4, 5, 7)$ with a MUX



as *one-hot encoding* provides for exactly one select line to be logic-1 per input. So instead of having $\log_2$ n number of select lines, where n is the number of inputs, we'd have n select lines. It turns out that in some senses this can be a desirable tradeoff. It can also easily give us things like 3:1 MUXes that might seem strange in the base-2 paradigms we typically use.

We can do other useful things with MUXes, too, such as implementing logic functions directly, completely bypassing any need to synthesize into equations! If we connect the potential values of our function (the truth table) up to the inputs of a MUX and the inputs to the function itself up to the select lines of the MUX, we have a device that will implement the function!

Take $F = \sum m(03, 4, 5, 7)$ for example. It's MUX implementation would be as shown in Fig. 12.6.

We can even get fancy and use smaller MUX's to build larger ones and implement the function like shown in Fig. 12.7.

This won't get you anywhere near the minimal representation, but sometimes it can be convenient.

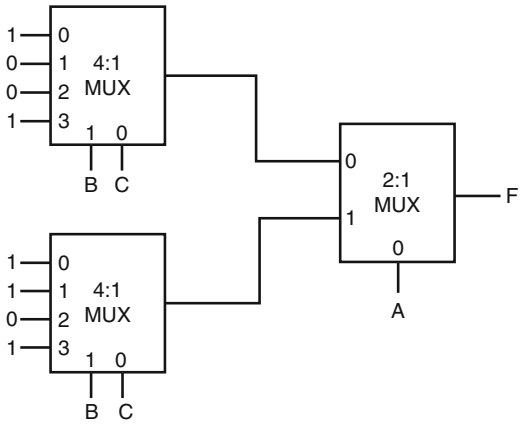


Fig. 12.7 Cascading multiplexers (what can't they do?)

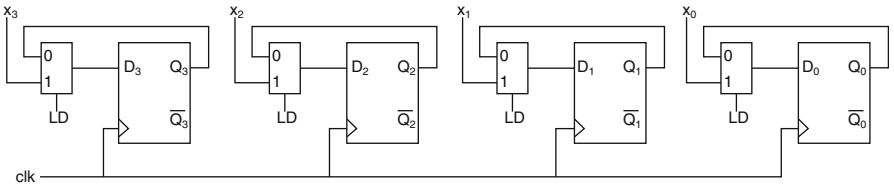


Fig. 12.8 4-bit register with LD signal implemented using MUXes

We can also use MUXes to build more sophisticated registers. In the previous chapter we looked at registers with LD signals using SR-Latches. Since we're going to be using mostly DFF's from here on out, we need to build a register with those. The issue is that the DFF doesn't have native hold functionality. We can use the MUX to build that in. In essence, the LD signal creates two options for the register input: either update with the new x being presented to the latch or updated with the current value already stored in the latch. Since we're facing a situation where we need to select one option out of a menu of options, we can utilize a MUX. Here, in Fig. 12.8, is the four-bit register with LD signal implemented using MUXes:

The LD signal serves as the select input to the MUXes. When LD = 1 we've designed it so that the new input x gets loaded into the DFF's. When LD = 0 the MUX selects the line attached to the Q's and simply maintains its current value. Pretty slick. We can use this concept to increase the number of interesting things our registers can do.

A **shift register** is a tremendously important digital design component. Shifting a binary word one or more bits in a certain direction is useful in algorithms ranging from signal processing to high-speed multiplication and division. It is also needed to play Super Mario Brothers: the original 8-bit Nintendo Entertainment system used a shift register in the controller to transmit the eight possible controller configurations to the console one bit at a time. So, this thing is quite versatile. Let's design it.

Table 12.1 Action table for specified register

S1	S0	Function
0	**0**	Hold
0	**1**	Load input x
1	**0**	Shift right
1	**1**	Shift left

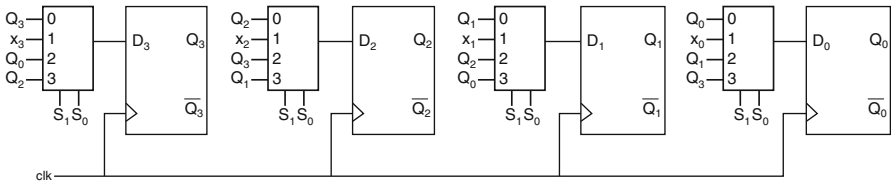


Fig. 12.9 Shift register of Table 12.1 implemented with MUXes

We need to build a register that can shift one bit to the left or one bit to the right. Thinking through the design, all we're doing is adding two more potential options to our current build. We can already load a new value into the register or maintain the current value. To shift we just need to also be able to load in the value from the DFF immediately to the right or left. Since we have four options, we'll need a 4:1 MUX which requires two control signals. Let's put all these thoughts from our head into a convenient table (Table 12.1).

We can now build the shift register (Fig. 12.9):

Instead of trying to draw all the wires and getting caught up in layout issues, we can simply label the inputs to the MUXes that are needed. Just make sure the control table for the register matches the inputs to the MUX and the register almost builds itself. We can even add things like shifting more than one bit, clearing, setting to a preset value, etc. It's just a matter of connecting different inputs to the MUXes feeding into the DFF's.

We could also add more descriptive signals such as SHL for shift left and SHR for shift right. Then, along with LD, we have three signals to choose among four options. We can design a 4:1 MUX with three select lines, and depending on the design constraints it maybe worthwhile to do so. Our designs in this text will rely on the basic power-of-2 based MUXes. (Things like 3:1 and 7:1 MUXes are sometimes used in digital designs as well, so it's not unheard of to bring these sorts of devices to the table.)

When we load the input x the way we did in the previous two examples, that is, all the bits at one time, we call it **parallel load**. We can also make a **serial in/out register** for doing important things like sending bits to the computer so it can tell you the princess is in yet another castle.

For a serial in/out register (Fig. 12.10), we output the values from the rightmost bit and input only into the leftmost bit. Its control input would be either to hold or accept on serial input.

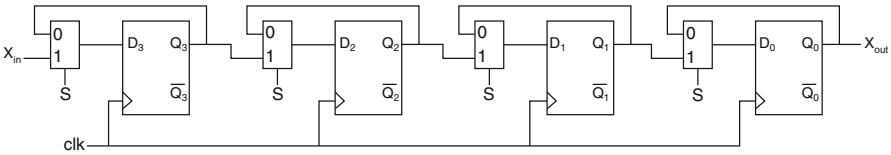


Fig. 12.10 Serial shift register using MUXes

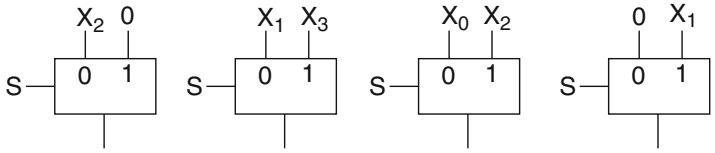


Fig. 12.11 Simple shift register framework

We can build registers with any sort of functions we need, and we'll feel free to just add control signals to our block diagram for the register without giving the DFF-level design each time.

Shifting is such an oft-used operation that we can do it outside the context of a register as well. Sometimes you don't want to take the clock cycles necessary to store a value and save the shifted version. Remember we have to wait for the rising edge of the clock in order to save values into a DFF. This means we can only perform one register-based operation per clock cycle and we can't then send it onward to other components in the same cycle we perform the shift. If we want to shift as part of a larger operation within a single clock cycle we can use a network of multiplexers without DFF's to accomplish this.

Consider a device that can shift one bit to the left or the right. We can just use the same principles from above but skip the part where we load into the DFF's.

In the design in Fig. 12.11, S = 0 means to shift left and S = 1 shifts right. It's easy to expand to any number of shifts in any combination that's needed. For example, consider the problem of doing something like dividing. If we want to multiply by a power of two, we can shift a number of bits equal to the exponent. (Right? Remember our binary? It's like when we multiply 38 by 100 and get 3800 because $100 = 10^2$.) What if we want to multiply by a non-power of 2? What if we need to multiply by a hideous number such as 54? While you can build a high-speed multiplier that can handle any computation, if we have some algorithm dependent upon multiplying by the constant 54 it may be worth a dedicated component. Figure 12.12 shows how we can build it with MUX-based shifters by noting that 54 = 32 + 16 + 4 + 2 and adding together the four shifted values.

The notation "n>>" means to shift right n, whereas "n<<" would mean to shift left n. (It's a C thing.)

Crucially, from the perspective of computing, is that we can see these multiplexors being useful in the datapaths for many of our instructions.

Now we can implement more sophisticated instructions such as BRz R1, R2, which will updated the value in the PC to PC = PC + R2 in the event the value in register R1 is zero.

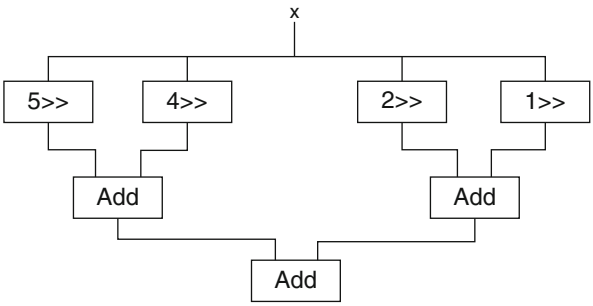


Fig. 12.12 Using shifting via MUXes to multiply by 54

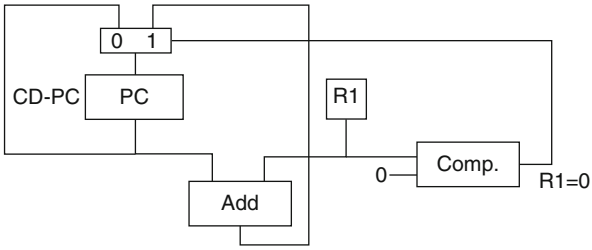


Fig. 12.13 Datapath multiplexing the input to the PC register

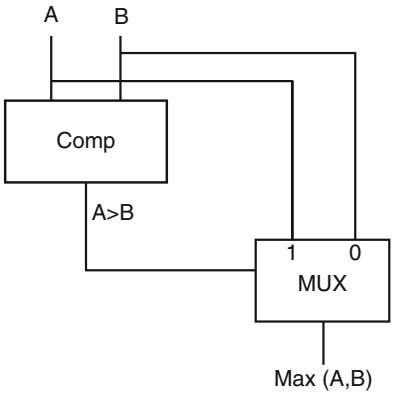


Fig. 12.14 Datapath using multiplexers and comparators to comput max (A,B)

Our datapath needs to be able to perform both the addition and the comparison. We can use the result of the comparison to drive a MUX choosing what value to load into the PC.

We're updating the value in the PC either with the value of PC + R1 or with the current value of the PC (so that there is no change.) We can use the magnitude comparator hardwired to compare vs. 0 to provide the control signal to a 2:1 MUX which determines what signal gets loaded into the PC. Figure 12.13 shows this.

We can also multiplexors and comparators together to choose which of two inputs, A or B, is larger and output it, as seen in Fig. 12.14.

With these components now in our toolbox, we're getting closer to being able to implement actual algorithms. And that, really, is the essential thing about digital design.

Exercises

12.1 Implement the following functions on 8:1 MUXes and on 4:1 MUXes:

(a) $F(A,B,C) = \sum m(1,2,5,6,7)$
(b) $F(A,B,C) = \prod M(0,1,4,7)$

12.2 Implement the following functions on 16:1 MUXes and on 8:1 MUXes

(a) $F(A,B,C,D) = \sum m(1,5,6,7,11,12,14)$
(b) $F(A,B,C,D) = \prod M(3,4,7,8,10,12,13,14)$

12.3 Implement the function $F(A,B,C) = \sum m(2,3,4,6)$ on a 4:1 MUX. However, reduce the variable A instead of the variable C. That is, make B and C the control inputs to the MUX.

12.4 Implement the function $F(A,B,C) = \prod M(2,3,5,6)$ using a combination of 4:1 and 2:1 MUXes.

12.5 Implement the function $F(A,B,C,D) = \sum m(0,1,3,5,6,8,10,11,14)$ using only 2:1 MUXes.

12.6 Design a 4-bit register using positive edge-triggered DFF's that takes a two bit control signal $s = s_1 s_0$ that works as specified in Table 12.2.

12.7 Design a 4-bit register using negative edge-triggered DFF's that takes a two bit control signal $s = s_1 s_0$ that works as specified in Table 12.3.

12.8 Design a 4-bit register using positive edge-triggered DFF's that takes two control signals LD and CLR. The register should update its value (parallel load) when LD = 1 and clear its value when CLR = 1. Give CLR priority over LD (so that when both signals are 1 the register clears instead of loads.)

Table 12.2 Action table for exercise 12.6.

s1	s0	
0	**0**	Hold
0	**1**	Parallel load X
1	**0**	Set all bits to 1
1	**1**	Complement all bits

Table 12.3 Action table for exercise 12.6

s1	s0	
0	0	Hold
0	1	Parallel load X
1	0	Reverse current value
1	1	Swap first two bits with last two bits

12.9 Design a 4-bit register using positive edge-triggered DFF's that takes two control signals LD and SHR. The register should update its value (parallel load) when LD = 1 and shift right when SHR = 1. Include a serial output as well as a parallel output on the register and give SHR priority over LD (that is, the register shifts instead of loads when both control signals are 1.)

12.10 Design a device which will determine which of three unsigned inputs A, B, and C is the largest and output that number. You may use MUXes, comparators, and logic gates in your design.

12.11 Design a device which will determine if three unsigned inputs A, B, and C are equal. It will output a signal E = 1 if all three are equal and a signal E = 0 otherwise. You may use comparators and logic gates in your design.

12.12 Design a device which will input two unsigned numbers, subtract the smaller of the two numbers from the larger, and output the difference. You may use MUXes, comparators, and a single adder/subtractor in your design.

12.13 Design a device which will output a 1 if an unsigned input X is between 50 and 100, inclusive, and output a 0 otherwise. You may use MUXes, comparators, and logic gates in your design.

12.14 Design a datapath which inputs three unsigned numbers x, y, and z, and outputs $x - y$ if $x - y > z$ and outputs $x + y$ otherwise. You may use MUXes, comparators, adder/subtractors, and logic gates in your design

12.15 Consider a logic function F that is equal to 1 for digits in today's date (4-19-2005) and 0 otherwise. Assume inputs in BCD. Implement this function using an 8:1 MUX.

12.16 Label the inputs to each of the four identical multiplexers in Fig. 12.15 so that the design below will behave as a special ring counter that works per the given function table. Label the parallel inputs I_3, I_2, I_1, I_0.

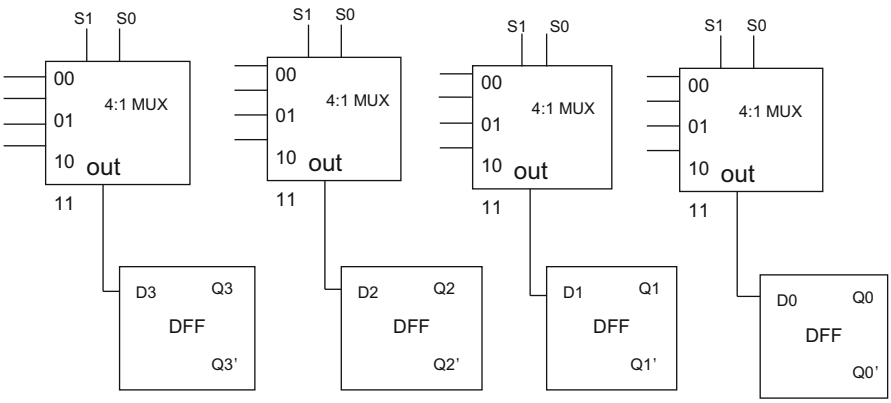


Fig. 12.15 Circuit diagram for exercise 12.16

S_1	S_0	Action
0	0	Load parallel inputs
0	1	Shift right
1	0	Shift left
1	1	Toggle

12.17 Implement the 4-variable logic function $F = \prod(0, 2, 3, 6, 11, 15)$ using a MUX. Your solution should consist of the higher-order device diagram, truth table, and logic diagram for the appropriately sized MUX as well as a second diagram showing the values connected to each input and select pin needed to implement the given function F. Be sure to clearly label the input pin numbers and the select pin numbers.

12.18 Design a barrel shifter that will shift a 4-bit input right or left depending on a 3-bit control signal $S_2S_1S_0$ where $S_2 = 1$ for a left shift, $S_2 = 0$ for a right shift, and S_1S_0 is a binary number from 0 to 3 indicating the magnitude of the shift.

There are two ways to design this device. You can use a lot of 4:1 MUXes or you can design a filter to preprocess the control signal in a certain way so that your problem reduces to the simple barrel shifter that always shifts left.

Your design should incorporate only MUXes and, if you so choose, a control filter (which must be fully described using combinational logic.)

12.19 Implement an 8:1 DMUX using only 2:1 DMUXes. Draw the device diagram for the 8:1 DMUX and be sure the signal names match up properly on your diagram with the 2:1 DMUXes.

12.20 Consider a 4-variable logic function F which is equal to 1 when the binary value of the inputs is equal to a digit in today's date (0, 1, 2, 6, or 7). Implement this function using a 16-to-1 MUX. You do not need to show the truth table or a gate level implementation. Just draw the device diagram for the MUX, label the input pins, and show what inputs connect to which pins.

12.21 Design a wrap-around shifter device. It should take a four-bit input and a two-bit control signal and output a shifted version of the input. Use the following control scheme: 00 = no shift, 01 = shift right 1, 10 = shift right 2, 11 = shift left 1. Wrap a bit around to the other side when necessary. For example, if the input is 1000, then shifting 1 right results in an output of 0100 and shifting 1 left results in an output of 0001.Use 4:1 multiplexers to implement this design.

12.22 Show how to construct a 16:1 multiplexer using only 4:1 multiplexers.

12.23 Show how to connect the following 8:1 MUX to implement the logic function $f(a, b, c) = \sum m(1, 2, 4, 7)$.

12.24 Show how to use two 4:1 MUXes and one 2:1 MUX to implement the logic function in exercise 12.23 above.

12.25 Show how to use variable reduction to connect an 8:1 MUX to implement the four-variable logic function $f(a, b, c, d) = \sum m(1, 3, 4, 7, 12, 13, 14)$.

12.26 Design a one-bit comparator that will take two input bits, $x$and$y$, and output two signals, $x = y$ and $x > y$. (You may name these signals E and G, respectively). Fill in the truth table, find the minimal sum logic, and draw the gate-level implementation for the device.

12.27 Design a comparator that will take two 3-bit inputs $x = x_2x_1x_0$ and $y = y_2y_1y_0$ and output $x = y$ and $x > y$signals (again, you may label these signals E and G, respectively). Use your one-bit comparator from Problem 7 along with logic gates in your design.

12.28 Design a device which will input two 4-bit unsigned numbers and subtract twice the smaller of the two numbers from the larger of the two numbers. (That is, input x and y and output $x - 2y$ if $x > y$ and output $y - 2x$ otherwise.) You may use any of the devices we've studied (n-bit adder/subtracters, n-bit comparators, decoders, and MUXes) as well as combinational gates in your design.

12.29 Design an arithmetic shifter. The device should input a 4-bit value $x$and a 2-bit control $c$and output a 4-bit value representing the shifted input. If c= 00 then shift left one, if c= 01 then shift right one, if c= 10 then shift right two, and when c= 11 output $x$without a shift.

12.30 For an example of how this works, consider the possible inputs $x_1 = 1011$ and $x_2 = 0111$. When we shift left we fill in with 0's, so that a shift of one left gives $x_1 = 0110$ and $x_2 = 1110$. When we shift right we fill with the sign bit, so that a shift of one right gives $x_1 = 1101$ and $x_2 = 0011$ and a shift of two right gives $x_1 = 1110$ and $x_2 = 0001$. You may use any of our datapath components, but it's easiest to do this using only MUXes: one for each output bit. Think about it for a while but don't overthink it—the design is very simple once you figure it out.

12.31 Show how to implement the same function $f(a, b, c, d) = \sum m(0, 2, 5, 11, 12)$ using an 8:1 MUX and a single NOT gate.

12.32 Implement the function $\mathrm{F(A, B, C, D)} = \mathrm{A} \bigoplus \mathrm{B} \bigoplus \mathrm{C} \bigoplus \mathrm{D}$ using 8:1 multiplexers and any additional logic gates needed.

12.33 Implement the function $f(x, y, z) = \sum m(0, 3, 4)$ using an 8-to-1 MUX.

12.34 Implement the function $f(w, x, y, z) = \sum m(0, 1, 5, 10, 13, 14)$ using an 8-to-1 MUX and external gates.

12.35 Implement the function $f(x, y, z) = \sum m(2, 5, 7)$ using a 4-to-1 MUX and external gates.

12.36 Implement the function $f(w, x, y, z) = \sum m(0, 1, 4, 9, 11, 15)$ using two 8-to-1 MUXes and a 2-to-1 MUX.

12.37 Show how to use variable reduction to connect an 8:1 MUX to implement the four-variable logic function $f(a, b, c, d) = \sum m(2, 3, 5, 8, 9, 10, 12, 15)$.

12.38 Design a 4-bit register using positive edge-triggered DFF's that takes a two bit control signal $s = s_1\ s_0$ that works according to the table given in Table 12.4.

12.39 Design an 8-bit register using positive edge-triggered DFF's. Include LD and CLR signals. The register should only update its value when LD = 1 and the register should clear its value (that is, make all bits 0) when CLR = 1.

Table 12.4 Action table for exercise 12.38

s_1	s_0	
0	0	Clear the register
0	1	Shift right with serial out
1	0	Parallel load an input X
1	1	Hold

12.40 Design a 4-bit register with a two-bit control input s_1s_0. When $s_1s_0 = 00$, then the register should maintain its current value. When $s_1s_0 = 01$, then the register should load the 4-bit input x. When $s_1s_0 = 10$, then the register should set all its bits to 1. When $s_1s_0 = 11$, then the register should complement each bit. Use positive edge-triggered DFF's.

12.41 Design a 4-bit register with a two-bit control input s_1s_0. When $s_1s_0 = 00$, the register should maintain its current value. When $s_1s_0 = 01$, the register should load the 4-bit input x. When $s_1s_0 = 10$, the register should reverse its current value (so that the LSB becomes the MSB, etc.). When $s_1s_0 = 11$, swap the most significant two bits with the least significant two bits (so that $x_3x_2x_1x_0$ becomes $x_1x_0x_3x_2$.) Use negative edge-triggered DFF's.

12.42 Use positive edge-triggered DFF's to design an 8-bit right shift register. It should feature parallel load when the LD input is 1 and serial out when the SHR input is 1 (that is, when SHR = 1 the value stored in the register should shift one to the right and the LSB should be output.)

Chapter 13
Decoders and Register Files

The multiplexer lets us select one of a number of inputs to pass through to the next device. The **decoder** allows us to select one of a number of connected devices to activate in response to an input (the eponymous *code*). These devices are similar and are often confused by those new to digital design.

One of the most important devices to be driven by a decoder is a register. Often in our designs we need to use many registers. Since each register has control signals such as LD associated with it, the more registers in a system the more of these signals need to be tracked. In order to keep this from getting unwieldy, we can organize registers into groups called a **register file**.

Register File Design

Figure 13.1 presents a first take on a register file of 8 registers:

First we see the 3:8 decoder (pronounced "3-to-8 decoder." It takes a three-bit input and activates one of the 8 output lines in response. This is an **active-high decoder** so when the output line is activated it is logic-1 and the inactive lines are logic-0. You can also have an **active-low decoder** symbolized using inversion bubbles on each of the outputs.

Each output line is attached to the LD signal for one of the registers. In this way we can replace the 8 individual LD signals for each of the registers with a single 3-bit input to the decoder.

There are still two problems: (1) since one output line of the decoder must always be active, we are forced to update one of the registers each clock cycle and (2) how do we deal with the 8 separate outputs from all these registers?

We can solve the first problem by adding a control signal to the decoder to **enable** or **disable** it. When the decoder is disabled then no output line is active. Therefore, we won't be required to change the value in a register every clock cycle. This is good.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_13

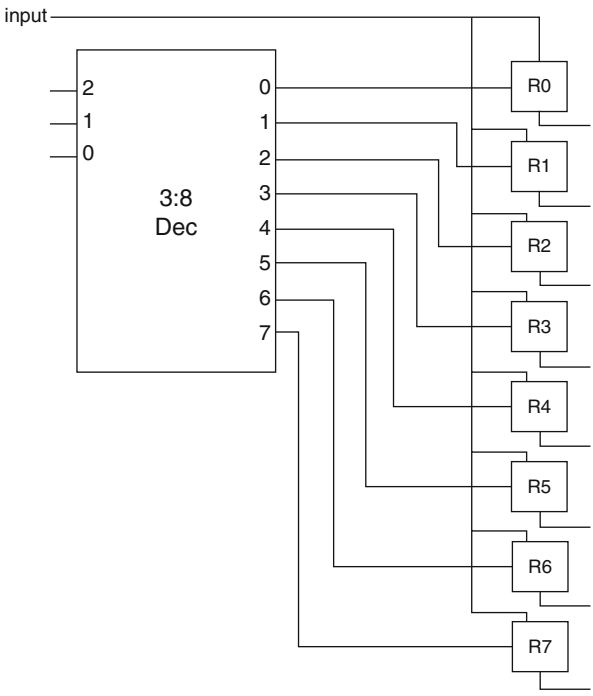


Fig. 13.1 First take on register file

We can solve the second problem by reusing our input "code" at the output side of the register file. A single 8:1 MUX can select one of the registers to output from the file. We now have the circuit of Fig. 13.2.

We call the three-bit input which selects the register sel and we call the enable LD_{RN} (when LD_{RN}= 1 we update a register and when $LD_{RN} = 0$ we maintain all values.) Notice sel controls both the 3:8 decoder as well as the 8:1 MUX. Putting this all together we get a single block diagram for a register file device (Fig. 13.3).

This device will be used in most of our datapath designs going forward.

Multi-port Register File

While the register file just detailed is quite useful, we're actually going to be wanting slightly more complex devices when it comes to implementing our computer datapaths. Recall from the Encoding Code chapter that we often want to source multiple operands for a single instruction. Using the decoder from the previous section would require a datapath like the one in Fig. 13.4.

Here we require a buffer register Y to hold the first output of the register file and another buffer register Z to store the output of the ALU. To execute an instruction like ADD Ra, Rb, Rb, we have to run through multiple steps (called *micro-operations*):

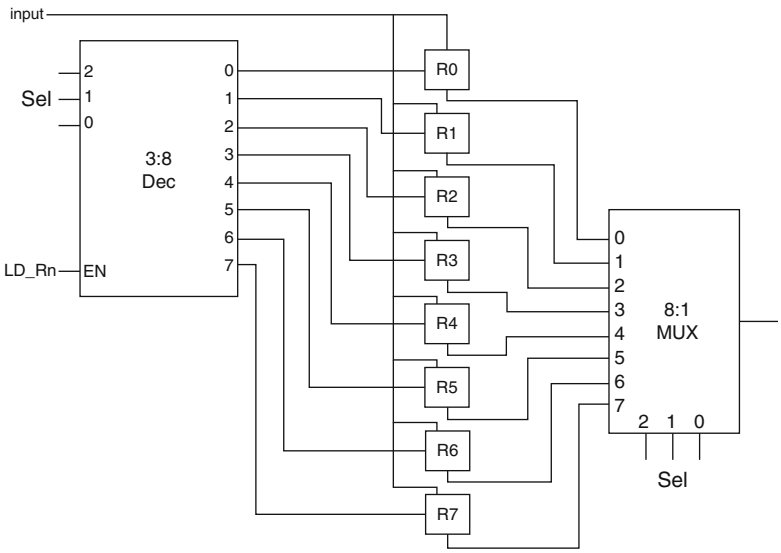


Fig. 13.2 A better register file design

Fig. 13.3 Register file block diagram

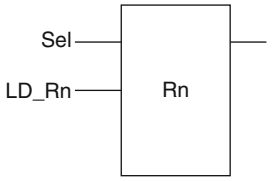
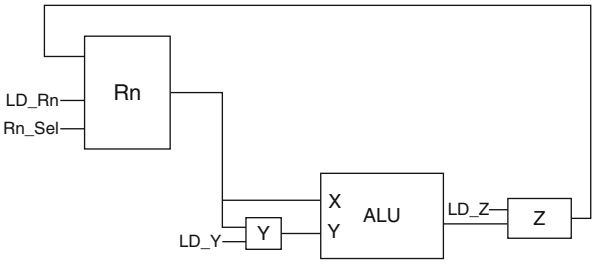


Fig. 13.4 Datapath using a register file and MUX to source multiple operands to the ALU



1. copy the value from one source operand to the buffer Y
2. perform the computation and store the result in Z
3. store the result from Z back into the register file

Why all the extra steps? They are necessary because of the way we designed our register file. With only one select input, this device is limited to outputing, or *reading*, exactly one register and also to updating, or *writing*, exactly one of its

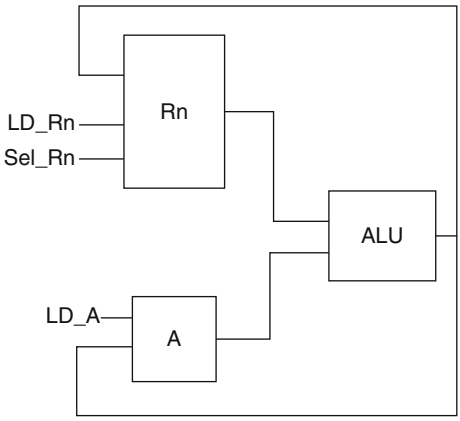


Fig. 13.5 Accumulator based datapath

registers. Furthermore, the register read and the register written also have to be the same!

In fact, devices like this actually limited the development of instruction sets. For a long time, computers accommodated register files of this kind with the inclusion of a special register called the **accumulator (A).** This accumulator took the place of the two buffers Y and Z shown in the diagram above. A typical accumulator-based design is given in Fig. 13.5.

In this datapath, the accumulator A is an input to and an output of every ALU operation. The only reason the ALU output connects to the register file Rn is because a simple data movement instruction like MOV Ra, A, which copies the value from A into a register in the file, actually goes through the ALU in this design. We could have optionally simply connected the output of A to the register file's input directly, but for larger system reasons we tend not to do that. The layout is cleaner if we imbue our ALU with a "pass through" operation which permits one of its input lines to continue on without effect.

In these architectures, in order to produce the results of a three-operand instruction such as ADD R1, R2, R3, which adds the values in R2 and R3 and stores the sum in R1, we have to process the three instruction sequence given by

```
MOV A, R2     A = the value in R2
ADD A, R3     A = the sum of the values in A and the value in R3
MOV R1, A     R1 = the value in A (which is now R2 + R3)
```

These correspond to the three steps illustrated above, with the accumulator A playing the role of both the Y and Z buffer registers.

Something vitally important has been observed here: **the language we use to express computation** (the instructions we have to sequence) **is directly affected by the underlying technology available for its implementation** (the quality of the register file in this case.) So, while you may not care about transistors or the nuances

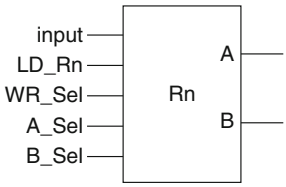


Fig. 13.6 Multi-port register file block diagram

of electromagnetic theory, they very much care about *the very thoughts in your mind* (at least as far as writing computer programs is concerned)!

OK, moving on to the design point: we would very much like to have access to a more sophisticated sort of register file to use in our digital systems. How sophisticated? In principle, we could design one that could write three registers and output five or whatever combination we need. In practice, the three-operand instruction proves optimal for most general-purpose applications so we'll focus on the design of the register file that best supports something like our ADD Ra, Rb, Rc instruction from above. What we need is the ability to read from two registers while simultaneously writing a third. We also want to ensure that one of the registers read could double as the one written (or even that the same register could be used for all three operations.) After all, our generic instruction ADD Ra, Rb, Rc can be instantiated as ADD R1, R2, R2 or even ADD R2, R2, R2.

To this end, let's catalog the control signals we're going to need. Only one load signal is necessary, along with a select signal indicating which register to modify. Two other select signals are mandatory in order to read from two registers. We call each potential output a **port** in the register file, and the final device a **multi-port register file.** Figure 13.6 shows our block diagram.

We have a single input line supported by LD_Rn and WR_sel controls. The WR stands for *write* and sometimes you'll see the LD signal in these cases called something like we or we_en which means "write enable." It's good to get a feel for the language of digital design so you can look at multiple sources or data sheets and still understand what the components do. The other two inputs are select lines for the A and B ports. Be careful not to confuse the port names A and B with the operand names in our three operand instructions like ADD Ra, Rb, Rc. The source operand Rb and the register file port B have nothing to do with each other: in principle, any of the encoded operands can be tied to any of the register file ports.

Let's put this all together in Fig. 13.7 into the full design for the dual-port register file.

This is the beginning of the full-fledged memory design explored in more detail in the Memory chapter. For now, however, let's investigate the design of the decoders at the heart of these devices.

Decoder Design

The decoder itself can be implemented with gates. We'll show the 2:4 version in Fig. 13.8.

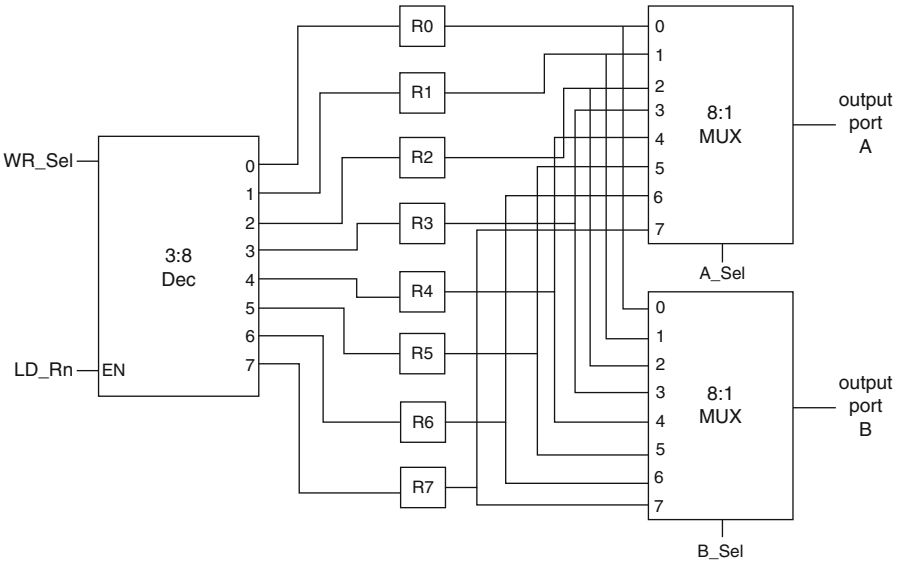
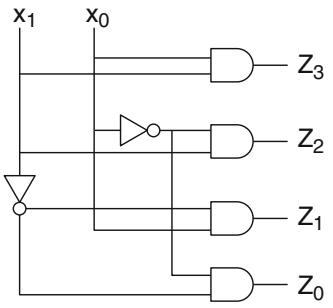


Fig. 13.7 Design of dual-port register file

Fig. 13.8 2:4 Decoder gate diagram

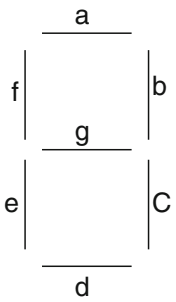


We can build 2:4 or 3:8 or 4:16 or any n:2^n decoders pretty readily. We can also make 3:5 or 2:7 decoders if we needed some for a particular purpose.

It's quite common to need a **function-specific decoder** to drive a particular system. In the Other Digital Representations chapter, we introduced the concept of a seven-segment display and discussed the use of the BCD encoding scheme in systems featuring such displays. Here we will show the steps involved in the design of a decoder for the seven-segment display.

Recall the display and the inputs that turn on each of the segments (Fig. 13.9).

We can now synthesize the functions and construct the gate-level diagram for this decoder. In this case, we have a 4-bit input code and a 7-bit output and we allow multiple output lines to be active at a single time.



Decimal	x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
	1	0	1	0	d	d	d	d	d	d	d
	1	0	1	1	d	d	d	d	d	d	d
	1	1	0	0	d	d	d	d	d	d	d
	1	1	0	1	d	d	d	d	d	d	d
	1	1	1	0	d	d	d	d	d	d	d
	1	1	1	1	d	d	d	d	d	d	d

Fig. 13.9 Seven segment display decoder truth table

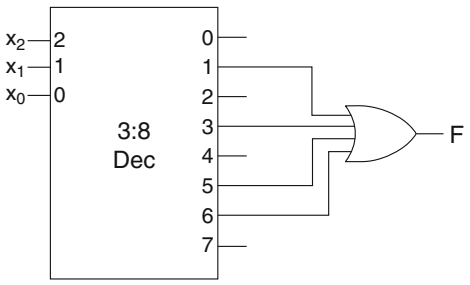


Fig. 13.10 Decoder implementation of $F = \sum(1, 3, 5, 6)$ using OR gates

Implementing Logic Functions with Decoders

Decoders can be used to implement logic functions. Consider $F = \sum(1, 3, 5, 6)$.

As seen in Fig. 13.10, we just connect the function's inputs to the decoder's select lines and then reason that F is logic-1 when the input is 001 or 011 or 101 or 110 and connect the appropriate outputs with an OR gate. Simple, but make sure you get this before continuing, because it's going to get more logically demanding.

Active-low decoders may also be used to implement this function.

Figure 13.11 shows that we now hook all the minterms to a NAND gate. Recall the NAND gate is 1 when any of its inputs are 0 (it's a low-level detector). An active-low decoder will set a minterm to 0 when it's activated by the input code. Therefore, we want F to be 1 when any of the outputs corresponding to minterms goes low. Thus, the NAND gate. (Again, it's easier to think through this when looking at the gates in this way. You could also do reasoning involving De Morgan's Law, but this is more straightforward for most people.)

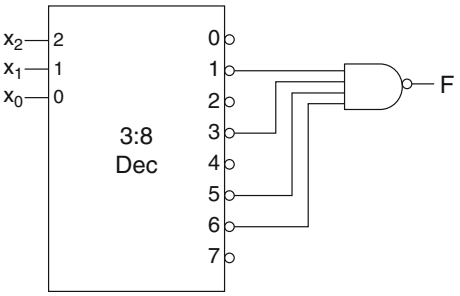


Fig. 13.11 Active-low decoder implementation of $F = \sum(1, 3, 5, 6)$

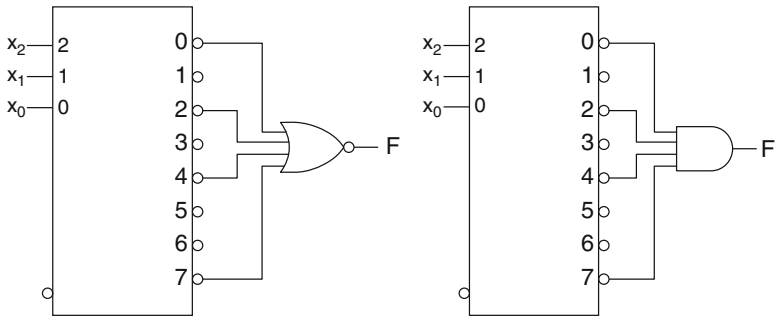


Fig. 13.12 Maxterm implementation of logic functions using decoders

It's also possible to implement the maxterm or POS version of a function using NOR and AND gates with active high and active low decoders, respectively. See Fig. 13.12.

Again, the logic is daunting unless you think of the gates as performing bitwise operations. The NOR gate emits a 0 so long as any of its inputs is 1 and is only 1 when all its inputs are 0. In the active-high decoder the maxterms are 1 when we want the function to be 0 and we only want the function to be 1 when all the maxterm outputs are 0. This is precisely the NOR gate. Similarly, the AND gate will be 1 when all its inputs are 1 which, for the active-low decoder, coincides with when none of the minterm outputs are active. Take some time to work through this, as the logic is not obvious. This is a great way to solidify your understanding of digital electronics.

Which implementation technique do we choose: minterm or maxterm? Typically we'll go with the one that requires the fewest inputs to tie into the gate. It's more efficient to work with 2 input NAND gates than 7 input AND gates, for example. This is why we need a strong grasp of the different forms our logic functions can take. Finding efficient implementations depends on it.

Like with MUXes, there are off-the-shelf components that include decoders for the express purpose of implementing functions. It's not going to give you the minmal gate layout, but its convenience is often very much worth it.

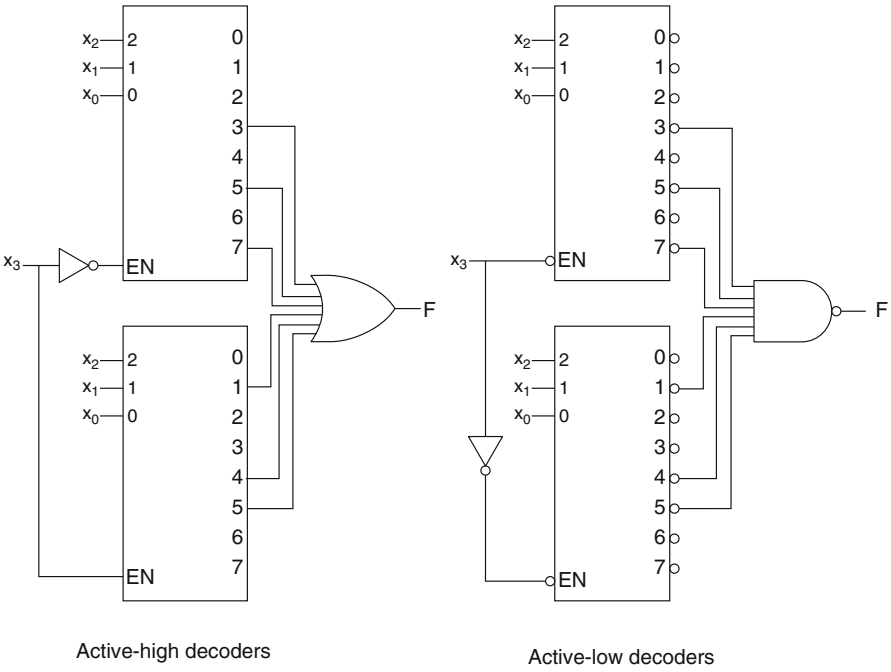


Fig. 13.13 Active high and active low cascading decoder implementation of $F = \sum(3, 5, 7, 9, 12, 15)$

It's also possible to build larger systems out of smaller decoders. We can use two 3:8 decoders to implement four variable functions. Consider the function $F = \sum(3, 5, 7, 9, 12, 15)$ and its implementation in Fig. 13.13 using both active-high and active-low decoders.

Notice that we send x_3 through an inverter in the active-high setup to make the high decoder the *low bank*—that is, it corresponds to the low half of the truth table where $x_3 = 0$. The lower decoder is the *high bank*. We can force other division as well. If we tied the least significant bit x_0 to the enables we'd have an *even bank* and an *odd bank* instead. You'd just need to be more careful to track which decoder outputs correspond to which minterms. Many systems are built using either of these configurations.

On the active-low circuit we have the inverter for x_3 on the bottom decoder instead of the top because of the way the active-low enable functions. Otherwise, the concepts are the same.

Encoders

The flip side of the decoder is the **encoder**, a device which detects activity on one of many input lines and converts that to activity on a combination of output lines. Where a decoder takes an input (the code) and selects exactly one output to activate,

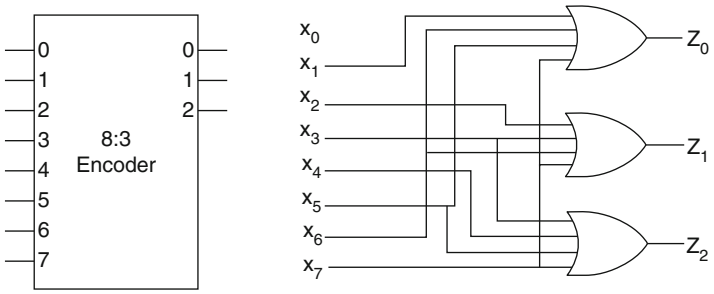


Fig. 13.14 Encoder block diagram and gate-level circuit

Table 13.1 Truth table for a priority encoder

X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	Z_2	Z_1	Z_0	valid
0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**
1	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**
d	1	0	0	0	0	0	0	**1**	**1**	**0**	**1**
d	d	1	0	0	0	0	0	**1**	**0**	**1**	**1**
d	d	d	1	0	0	0	0	**1**	**0**	**0**	**1**
d	d	d	d	1	0	0	0	**0**	**1**	**1**	**1**
d	d	d	d	d	1	0	0	**0**	**1**	**0**	**1**
d	d	d	d	d	d	1	0	**0**	**0**	**1**	**1**
d	d	d	d	d	d	d	1	**0**	**0**	**0**	**1**

an encoder responds to activity on exactly one of a number of input lines by outputting a combination of signals (the code).

In Fig. 13.14, we see the block diagram and gate-level design of an 8:3 encoder.

To understand the gate design, realize all we're doing is encoding the subscript of the input line being activated. For example, input x_6 outputs the code $z_2 = 1$, $z_1 = 1$, $z_0 = 0$ because 110 in binary is 6 in decimal. In the diagram, then, you can see the x_6 input tied to the OR gates for outputs z_2 and z_1. It's a bit awkward that input x_0 has nowhere to go, but that's how binary works. Sorry, x_0..

It's also important that exactly one input line is active at any time. Otherwise, we end up with the bitwise OR output of the input codes. That's typically not what we want. We can make an adjustment to the encoder design to accommodate multiple simultaneously active input lines. Instead of performing some bitwise operation on the input lines, what we usually want to do is encode only one of the available inputs. In order to decide which one to encode, we can assign priority levels to the various inputs. The resulting device is called a **priority encoder**.

The truth table for a 8:3 priority encoder is given in Table 13.1.

The priority encoder has a new output, *valid*, not present in the regular encoder. This signal differentiates between an absence of an input and the case where the input whose code is $z_2 = 0$, $z_1 = 0$, and $z_0 = 0$, because otherwise the two outputs

would be indistinguishable. This sort of a signal to indicate when output should be trusted and when it should be ignored is very common in digital components overall and we will use it often when we design complicated state machines later on. It's key in devices which have the capacity to have outputs which take time to stabilize on an output or, like in this case, have the potential to output the same values for distinct inputs (where the ambiguity is not desired.) We can see this in that the first and last rows of the truth table where the outputs differ only in the value of the *valid* bit.

Also note that the truth table is populated with don't cares! This is the **priority** element in play: we prioritize based on subscript where the lowest subscript has the highest priority. So, if both inputs x_3 and x_5 are active, the device only cares about the one with the lowest subscript, x_3. We can design priority encoders to implement any priority scheme we want.

Adding *valid* outputs to our normal encoders lets us build larger encoders. Consider the 16:4 encoder constructed from 4:2 encoder shown in Fig. 13.15.

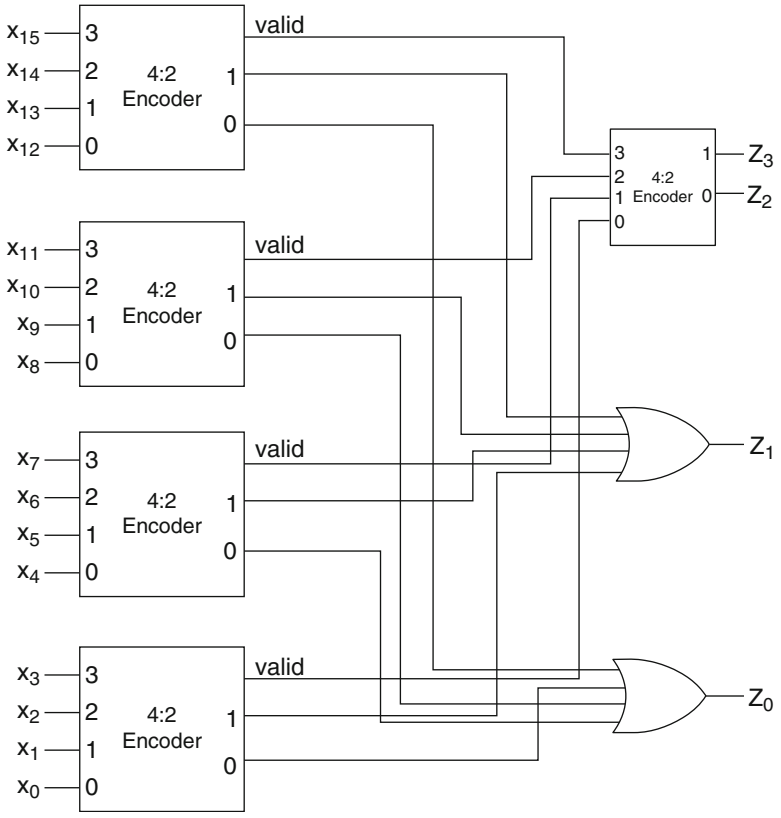


Fig. 13.15 Cascading encoders

The valid bits themselves encode the two most significant bits of the output while the two least significant bits are determined by ORing together the high and low outputs of the 4:2 encoders. This is not the easiest thing to work though and follow, so make sure it gels for you. This is an example of how being precise, having comfort with the underlying mathematics of how various elements in our digital systems are represented, and knowing the details of operation of all the individual components really helps in coming up with strong designs.

Exercises

13.1 Implement the function $F(A,B,C) = \sum m(1,3,5,6)$ using (a) an active-high decoder and a single OR gate, (b) an active-high decoder and a single NOR gate, (c) an active-low decoder and a single NAND gate, and (d) an active-low decoder and a single AND gate.

13.2 Implement the function $F(A,B,C) = \prod M(0,1,2,6)$ using (a) an active-high decoder and a single OR gate, (b) an active-high decoder and a single NOR gate, (c) an active-low decoder and a single NAND gate, and (d) an active-low decoder and a single AND gate.

13.3 Implement the function $F(A,B,C,D) = \sum m(0,3,5,7,10,11,12,15)$ using (a) two active-high 3:8 decoders and (b) two active-low 3:8 decoders.

13.4 Consider a system with three special devices which need to be enabled (with active high signals). Use decoders to interface to these devices such that device A will be enabled if the input code is less than or equal to 3, device B will be enabled if the input code is between 4 and 6, inclusive, and device C will be enabled if the input is 7.

13.5 Suppose you have eight active-low lights and eight active-high lights. You also have a switch and a dial. The switch can be in one of two positions and the dial can be in one of eight positions. Design a system using decoders that allows you to turn on one of the lights using the switch and the dial.

13.6 Let $F = \sum(0,1,2,7)$. Use a 3 to 8 decoder and a NAND gate to implement this function.

13.7 Design a three-bit two's complement decoder. Show the truth table and draw the gate-level implementation of the device. You should have one output line for each possible decimal equivalent in the three-bit two's complement representation.

13.8 Design a two bit priority encoder. You should have two output lines Y1 and Y0. Y1 should be 1 when the most significant input bit is 1 and Y0 should be 1 when the least significant input bit is 1. Give the most significant bit priority. Show the truth table for this device and draw the gate-level implementation.

13.9 Show how to construct a 4-to-16 binary decoder using only 3-to-8 binary decoders with enable inputs.

13.10 Show how to connect the following 3:8 decoder with active low outputs to implement both the logic functions $f(a, b, c) = \sum m(0, 1, 4, 7)$ and $g(a, b, c) = \prod M(0, 1, 2, 4, 6, 7)$.

13.11 Show how to connect two 3:8 decoders with active low outputs and active low enable signals to implement the four variable function $f(a, b, c, d) = \sum m(4, 7, 11, 14, 15)$.

13.12 Suppose a 3:8 decoder is attached to an array of 8Kx8 memory chips, the low 13 bits of the address access the locations within each memory chip, and the most significant 3 bits of the address are tied to the select inputs of the decoder (A_{15} is MSB, A_{13} is LSB.) Give the range of memory addresses (**in hex**) associated with the memory chip enabled by decoder output y_6.

13.13 Use the following **active-low** decoder and **NAND gates** to implement $f(a, b, c) = \prod M(1, 2, 3, 4)$.

13.14 Use the following **active-high** decoder and **NOR gates** to implement $f(a, b, c) = \sum m(2, 5, 6, 7)$.

13.15 Use the following **active-low** decoder and **AND gates** to implement $f(a, b, c) = \sum m(2, 4, 5)$.

13.16 Show how to connect two 3:8 decoders with active low outputs and active low enable signals to implement the four variable function $f(a, b, c, d) = \sum m(0, 2, 5, 11, 12)$. **You may use a single NOT gate and a single NAND gate in addition to the decoders.**

13.17 Modify the 3-to-8 decoder circuit below to implement the function f(a, b, c) = ac + bc.

13.18 Consider an array of 128Kx8 SRAM chips accessed by a 3-to-8 decoder. If the three most significant bits of the address are connected in order to the inputs of the decoder, what is the range of memory addresses (in hex) for the chip enabled by the $y_3$output?

13.19 Use a 3-to-8 decoder and OR gates to implement the logic function $f(w, x, y, z) = \sum m(1, 2, 3, 4, 11, 12)$.

13.20 Suppose you have 16 registers with active-low load signals. Design an interface for these registers that will take as input a single 4-bit number $x = x_3x_2x_1x_0$ and a single bit R.

The interface should have two parts: **write** and **read**. The write interface should, if the input R = 0, generate the load signal for the corresponding register based on the input x. If R = 1 then no load signals should be generated. The read interface should output the value stored in the register indicated by the input x. You may use 3:8 decoders, 8:1 MUXes, 2:1 MUXes, and logic gates in your design. You need only draw the interfaces; you do not need to draw the 16 registers.

13.21 Design a 3-to-8 Two's Complement Decoder using two 2-to-4 Two's Complement Decoders. The 2-to-4 Two's Complement Decoder should take a 2-bit Two's Complement number as input and assert one of 4 output lines corresponding to the appropriate decimal equivalent. This decoder should also have an Enable input line. The 3-to-8 decoder should incorporate two 2-to-4 decoders. Include truth table, output equations, combinational logic

diagram, and device diagram for the 2-to-4 decoder as well as the diagram for how you wire the 3-to-8 decoder in your solution. Be sure to clearly indicate which output lines correspond to which decimal numbers and which input lines correspond to which element of the 3-bit input signal.

Chapter 14
Counters

Sequential logic devices which increment, decrement, and otherwise manipulate and retain a **count** of something are called, simply, **counters.** In this chapter we'll look at several counters and see what uses different count sequences may have beyond simply keeping track of how many of something there are. Also, perhaps most importantly, this chapter will introduce a **sequential logic design paradigm** that we'll use throughout the rest of the book to design systems based on registers. From sequence detectors to computer control units, this process is going to allow us to design the most critical infrastructure in our most interesting systems. For that reason alone, it's a good idea to make sure you work through this chapter. As a bonus, we've got some interesting and important counters lined up for you. Enjoy!

Binary Counters

Each counter has a *count sequence*. That is, each counter runs through a particular sequence of numbers in a predetermined order. A **binary counter** proceeds along the path charted by the unsigned binary number representation. The 3-bit binary counter, for example, starts in 000 and then visits counts 001, 010, 011, 100, 101, 110, and 111 before finally returning back to 000 again. An **up counter** can count up while a **down counter** counts down. An **up/down counter** counts both up and down. Not rocket surgery, this.

A straightforward way of implementing a binary counter would be to attach an adder to a register. This design is highlighted in Fig. 14.1

The counter input *inc* activates the LD input on the register which updates the value stored to that value plus 1. If you want increment by 2 or 3 or 4 instead of by 1 you can hardcode in a different number as input to the adder. The problem with this design is that it's not efficient: the adder is an involved combinational circuit and we can try to do better by designing the necessary logic within the counter itself rather than relying on an external arithmetic unit.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_14

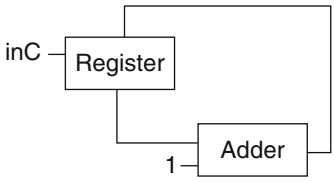


Fig. 14.1 Adder + Register counter design

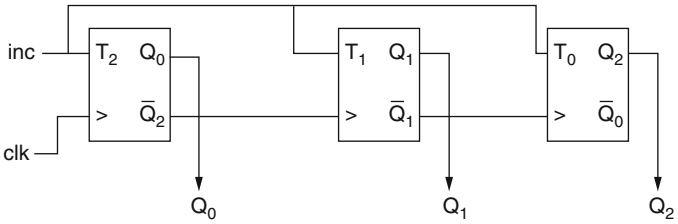


Fig. 14.2 Binary counter design using T flip flops

If we're clever, and we certainly are, we can note regularities in the binary count sequence. The least significant bit changes every time we increment. The next least significant bit changes every other time we increment. Bit position 2 change every other every other time we increment, and so on. We can then conclude that counting through the binary sequene can be achieved without any actual *adding* at all: if we "fip", or "toggle", the bits in the right orders we'll be good to go. If only we had a device that could toggle bits. . .

In Fig. 14.2 we see the binary counter design with Toggle flip flops. We use the complement of the previous bit position to control the timing of the next. This makes sure the bits only increment upon a change from 0 to 1 in the previous position. The increment input will always toggle the least significant bit, with each successively higher bit transitioning as appropriate mathematically.

While clever, this design process has no chance of generalizing to different count sequences. What we want is a counter design that is reasonably efficient along with a design process that we will extend to other *state machines*.

Our counters will store a value called a *count*. More generally, we can think of this value as the *state* of the system: it's a snapshot of everything important happening in the device. Our design process will be relevant for all state machines we work with going forward. In fact, we've already seen a hint of it in the Latches, Registers, and Timing chapter when we built generic AB Flop Flop devices.

So, here is how we'll think about counter design using this generalized design process. As we learned in the very beginning, the first step is to specify the logic functions needed by our design. In the case of a state machine, the logic functions are those that drive the correct transition from one state to another. We'll call them the *next state* logic functions and we'll call the register that holds the count the *state memory*.

Table 14.1 State table for the up/down binary counter

Present state			Input	Next state		
Q2	Q1	Q0	C	D2	D1	D0
0	0	0	0	1	1	1
			1	0	0	1
0	0	1	0	0	0	0
			1	0	1	0
0	1	0	0	0	0	1
			1	0	1	1
0	1	1	0	0	1	0
			1	1	0	0
1	0	0	0	0	1	1
			1	1	0	1
1	0	1	0	1	0	0
			1	1	1	0
1	1	0	0	1	0	1
			1	1	1	1
1	1	1	0	1	1	0
			1	0	0	0

In the case of implementing the counter with DFF's as the state memory element, we have the state table (Table 14.1) for our up/down binary counter.

We represent the current state with Q (we'll do this in all our state machine designs) and the next state with D (in this case because we're using DFF's.) The input c will count up when $c = 1$ and count down when $c = 0$. We wrap around from 000 to 111 and 111 to 000 when needed. Make sure you understand what information is present in this table, as the rest of this chapter and, ultimately, the entire chapter on State Machines is fully dependent upon it.

Once we've specified the logic functions that describe our system, the next step is to synthesize them. We'll find the minimal sum form and we'll use the variable-entry K-map method introduced in the Advanced Logic Function Minimization chapter. If you don't care about the minimization process, you can skip over this part and go straight to the implementation details. (But why would you deprive yourself of such awesome mathematics? Your call.)

In Fig. 14.3 we see the resulting K-maps and minimal sums for D_2, D_1, and D_0:

We have several options for how we group the cells in D_2, so if you get a different minimal form that's still potentially alright. Notice that the D_0 equation matches exactly our expectation from the Toggle flip flop implementation that the least significant bit would be constantly changing. Finally, we implement these equations using gates. Instead of drawing out each and every circuit, we'll group all the next state logic together as in Fig. 14.4.

As discussed in Chapter 13, we can also implement registers using SR, JK, or T flip flops. In these cases instead of the next state itself being the logic function we

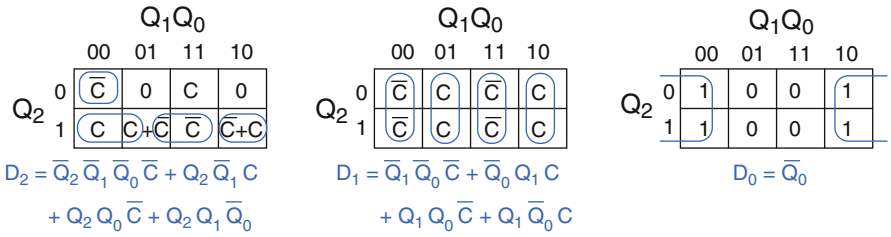


Fig. 14.3 K-maps for the counter of Table 14.1

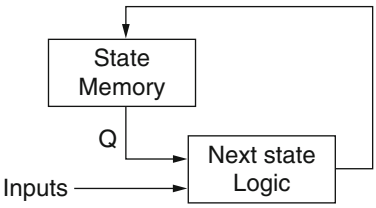


Fig. 14.4 Generic counter implementation diagram

need to synthesize, we need to specify implementation functions for each bit in the state. Refering back to the implementation tables for these flip flops we present the following table. Keep in mind this is just for the sake of completeness; we will definite not be implementing our counters every time with all these different flip flops! (Table 14.2)

As an example, in Fig. 15.5 we see the minimal sums for the SR implementation of this counter. The JK and T logic equations are left as an exercise for the reader (and it's really worth it to compare the JK with the SR, in particular, given the parallels in the impementation tables of the two flip flops.)

There we have it! We've designed a binary up/down counter and worked with a new design process. It's instructive to look at the regularities within the design that are brought out by study of the variable-entry K-maps in Fig. 14.5. This may be one of the best remaining uses for K-maps, actually, as we can see the abstract properties of certain systems visually in these maps in a way that's totally obscured by the use of synthesis software. So, for those of a more philosophical or mathematical inclination, I really do think getting a fluent grasp of the intricacies of the variable entry K-map is huge. It's not just a tool for computation that's eclipsed by automation. It can be a viewscreen laying bare the abstract structure at the heart of certain designs.

Now, let's have another go with a different kind of counter.

Table 14.2 Truth table for SR, JK, and T implementations

Present state			Input				Implementations													
Q2	Q1	Q0	C	Next state			S2	R2	S1	R1	S0	R0	J2	K2	J1	K1	J0	K0	T2	T1	T0
0	0	0	0	1	1	1	1	0	1	0	1	0	1	d	1	d	1	d	1	1	1
			1	0	0	1	0	d	0	d	1	0	0	d	0	d	1	d	0	0	1
0	0	1	0	0	0	0	0	d	0	d	0	1	0	d	0	d	d	1	0	0	1
			1	0	1	0	0	d	1	0	0	1	0	d	1	d	d	1	0	1	1
0	1	0	0	0	0	1	0	d	0	1	1	0	0	d	d	1	1	d	0	1	1
			1	0	1	1	0	d	d	0	1	0	0	d	d	0	1	d	0	0	1
0	1	1	0	0	1	0	0	d	d	0	0	1	0	d	d	0	d	1	0	0	1
			1	1	0	0	1	0	0	1	0	1	1	d	d	1	d	1	1	1	1
1	0	0	0	0	1	1	0	1	1	0	1	0	d	1	1	d	1	d	1	1	1
			1	1	0	1	d	0	0	d	1	0	d	0	0	d	1	d	0	0	1
1	0	1	0	1	0	0	d	0	0	d	0	1	d	0	0	d	d	1	0	0	1
			1	1	1	0	d	0	1	0	0	1	d	0	1	d	d	1	0	1	1
1	1	0	0	1	0	1	d	0	0	1	1	0	d	0	d	1	1	d	0	1	1
			1	1	1	1	d	0	d	0	1	0	d	0	d	0	1	d	0	0	1
1	1	1	0	1	1	0	d	0	d	0	0	1	d	0	d	0	d	1	0	0	1
			1	0	0	0	0	1	0	1	0	1	d	1	d	1	d	1	1	1	1

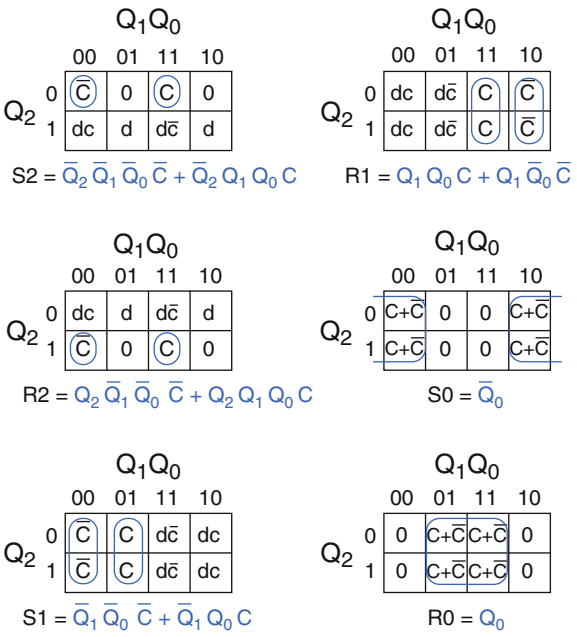


Fig. 14.5 K-maps for the SR flip flop next state logic specified in Table 14.2

Mod n Counters

A **mod n counter** is a binary counter with a count sequence of n numbers. Typically in a mod n counter n will not be a power of 2. As an example, we'll design a 3-bit mod 6 up counter. When the input c = 0 we will maintain our current value and when the input c = 1 we will increment to the next value. When we reach the end of our count sequence we will roll back over to 000.

In Table 14.3 is presented the truth table for this device, assuming DFF's for the state memory.

Since we want our mod 6 counter to have 6 count states, that means our sequence is from 0 to 5. Be careful here! It's natural upon first encountering this language to think mod 6 resets at 6, but really it resets at 6. Such is one of the joys at beginning our counts as 0. You should try to do this in your daily life for a while. It'll help you design counters.

States 110 and 111 are laden with don't cares because we don't ever expect our counter to actually be in those states.

Our next step is to synthesize the logic functions. Again, as Fig. 14.6 shows, we'll go for minimal sums using variable-entry K-maps.

These can be implemented with other flip flops for state memory, but we'll stop here for this one.

Table 14.3 Truth table for mod 6 up/down counter

Present state			Input	Next state		
Q2	Q1	Q0	C	D2	D1	D0
0	0	0	0	0	0	0
			1	0	0	1
0	0	1	0	0	0	1
			1	0	1	0
0	1	0	0	0	1	0
			1	0	1	1
0	1	1	0	0	1	1
			1	1	0	0
1	0	0	0	1	0	0
			1	1	0	1
1	0	1	0	1	0	1
			1	0	0	0
1	1	0	0	d	d	d
			1	d	d	d
1	1	1	0	d	d	d
			1	d	d	d

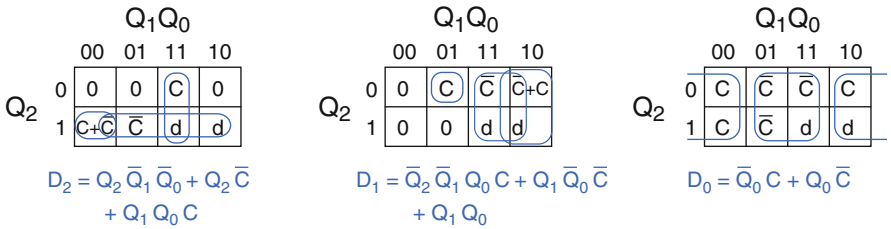


Fig. 14.6 K-maps for the counter specified in Table 14.3

Unit-Distance Counter

What about a non-binary count sequence? Recall the unit-distance encodings from the Other Digital Representations chapter: they are ones where successive numbers differ by only one bit. While the Gray code is a strong choice for counters with power-of-2 states, we're going to design a *3-bit mod-6 up/down unit-distance code counter*. Because we're not using a power of 2 the Gray code is not our best option here. We want the last number in our count to differ from the first number in our count by one bit, and the Gray code only ensures that happens for powers of 2 numbers. Therefore, we need to figure out a new count sequence.

We can actually use the K-map to help us here. Since the entire point of the K-map is that adjacent cells have one-bit differences, we can just plot out our six needed states such that the first and last are adjacent.

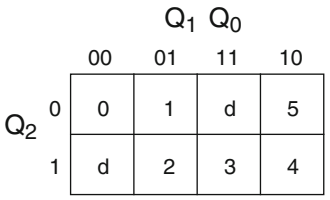


Fig. 14.7 K-map assisting design of the unit-distance counter

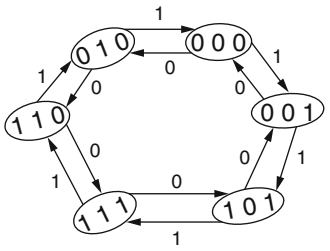


Fig. 14.8 State diagram for the up/down mod 6 counter

In the map in Fig. 14.7, we've assigned our state 0 the encoding 000, state 1 to 001, state 2 to 101, state 3 to 111, state 4 to 110, and state 5 to 010. We've maintained the one-bit difference between state 0 (000) and state 5 (010). States 100 and 011 are unused and marked on the map with don't cares. This path is not unique; others exist. You don't even have to start your fist count at 000! Start anywhere you want and it's still a unit-distance counter. Different count sequences have different applications and a specialized count sequence may be just what a particular design calls for. Since we've just constructed our sequence we are certainly not fluent in it yet, so it's helpful to represent the action of this counter pictorially with a **state diagram.**

In Fig. 14.8 we can see the digital representations associated with each state following the six numbers in our unit-distance code sequence: 000, 001, 101, 111, 110, and 010. The arrows represent transitions from one count/state to the next. The labels on the arrows indicate the value of the input that causes each transition: c = 1 counts up and c = 0 counts down.

In Table 14.4, we can see the unused states in the middle with the don't cares. We have to inspect each present state to find it's place in the overall count sequence. It's important to see here that even with a non-binary count sequence the truth table is still set up with the present states in binary order. We adjust on the specification side not on the ordering of the truth table. When first starting out with these designs, it's easy to get caught up in trying to reorder the table itself. Don't do this! The tables are fixed in this business and it's on us to make sure we're specifying the logic correctly.

In Fig. 14.9, we see the K-maps for each next state function:

Table 14.4 State table for the diagram of Fig. 14.8

Present state			Input	Next state		
Q2	Q1	Q0	C	D2	D1	D0
0	0	0	0	0	1	0
			1	0	0	1
0	0	1	0	0	0	0
			1	1	0	1
0	1	0	0	1	1	0
			1	0	0	0
0	1	1	0	d	d	d
			1	d	d	d
1	0	0	0	d	d	d
			1	d	d	d
1	0	1	0	0	0	1
			1	1	1	1
1	1	0	0	1	1	1
			1	0	1	0
1	1	1	0	1	0	1
			1	1	1	0

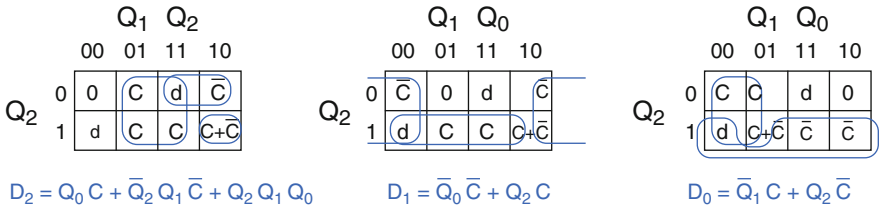


Fig. 14.9 K-maps for the state machine specified in Table 14.4

Ring Counters

A *standard* ring counter, also called a Johnson or Mobius counter, cycles through a count sequence featuring exactly one 1 in each bit position. An alternative ring counter called the *twisted* ring counter is similar but uses increasingly filling and then emptying 1's. These two types of sequences are shown in Fig. 14.10.

In Fig. 14.11 are collected the state diagram, state table, and next state equations for a 3-bit twisted ring counter. Note this counter doesn't have an input. We are designing it to increment every single clock cycle.

At the end of the day, it's pretty straightforward to implement our twisted ring counter. If we were inspired by the clever use of toggle flip flops in the creation of our first binary counter at the beginning of this chapter, it's possible that rather than crunch through the state table and K-maps to get these equations we'd have brewed

Fig. 14.10 Ring counter count sequences

Standard	Twisted
0000	0000
1000	1000
0100	1100
0010	1110
0001	1111
	0111
	0011
	0001

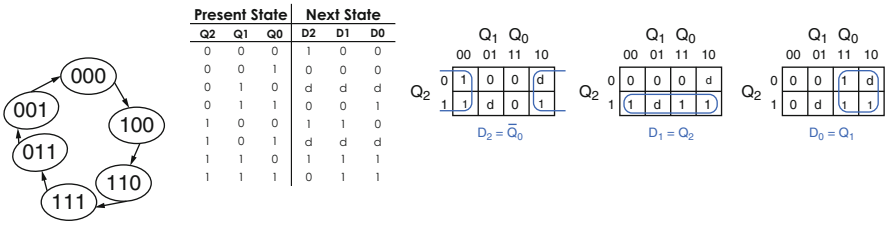


Fig. 14.11 State diagram, state table, and K-maps for the 3-bit twisted ring counter

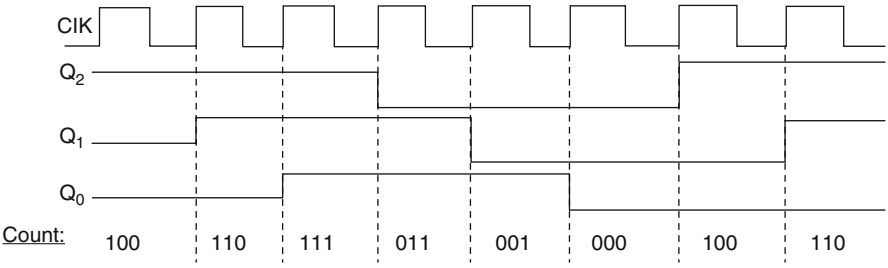


Fig. 14.12 Timing diagram output of the twisted 3-bit ring counter

up some hot beverage, settled under a tree in the park, and contemplated the connections needed for this device. They are so simple they could have been arrived at in this manner. But, the overall sequential logic design process helps us even when the resulting configuration is not so directly contemplatable.

What do we do with these ring counters? Why would we want to count in this fashion? It's important to keep in mind that our normal association with the word *count* betrays us here: our counters are not only for keeping track of the number of times something happens. Rather, the output of the counter can itself be the input to some other digital component. A count sequence, then, can be looked at as a special kind of output pattern.

In this light, it's instructive to look at the output generated by our ring counters. As sequential devices, the outputs are controlled by our clock synchronization signal. Figure 14.12 shows the relevant timing diagram for our 3-bit twisted ring counter:

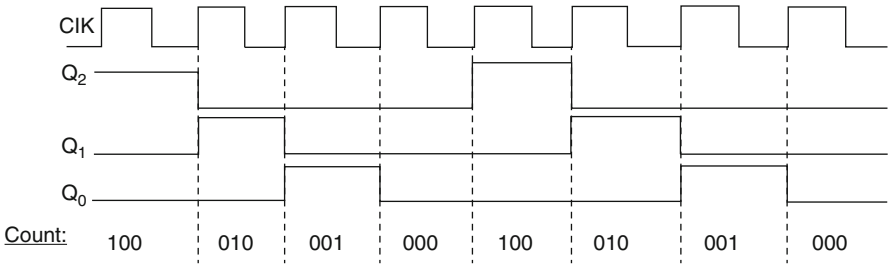


Fig. 14.13 Timing diagram of the standard 3-bit ring counter

For each bit of the count sequence, we get a square wave with a period of three clock cycles. Other devices can key off this counter for their own performance. These pulses are very useful in a lot of digital systems applications.

The standard 3-bit ring counter gives us the timing diagram in Fig. 14.13.

This diagram portrays a different sort of pulse. Each bit taken individually produce a single pulse and all together this counter outputs a continual pulse sequence. Other devices can rely on this sequence.

It's good for you to grasp that the count, or state, of our counter is only sometimes used to track occurrences of a thing. Often it is used to cycle through output patterns that are relevant elsewhere in the digital system. When we discuss state machines overall, we'll see that we can separate the state encoding from the output of the device. This gives us even more freedom to generate sequences of outputs to drive all sorts of other digital systems, even computers.

Exercises

14.1 Design a counter that counts from 3 to 9. The output should be the binary representation of the current count. Calculate the next state equations using DFF's. You will need one DFF for each bit in your state representation. The fewer number of bits you use in your state representation, the easier these calculations will be. Calculate the output equations (one for each output bit.)

14.2 Design a counter that runs through the count sequence in Table 14.5

14.3 You will need 3 DFF's to store the state information. Calculate the next state equations for each DFF and draw the ensuing state diagram (there are no output equations here.) Also check the unused states 101 and 010 and show where they fit within the state diagram.

14.4 Design a Ring/Johnson counter with an extra signal, "J." When J = 0, it will act like a Ring Counter and when J = 1, it will act like a Johnson Counter. You must use the least number of gates possible among the basic gates: AND, OR, NOT.

Table 14.5 Count sequence for exercise 14.2

State	Sequence
a	000
b	001
c	011
d	111
e	110
f	100
a	000

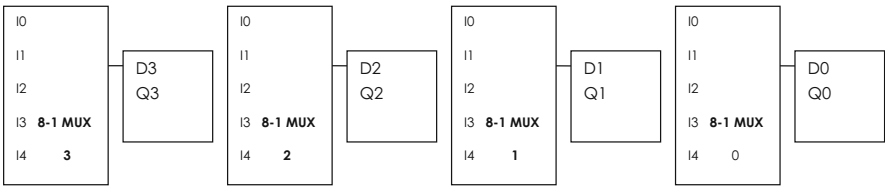


Fig. 14.14 Mux diagram for exercise 14.5

Table 14.6 Shift functions for exercise 14.5

Command			Function
S2	S1	S0	
0	0	0	Clear All
0	0	1	Hold All
0	1	0	Shift Right
0	1	1	Shift Right Twice
1	0	0	Shift Left
1	0	1	Shift Left Twice
1	1	0	Toggle All
1	1	1	Set All

14.5 Using the multiplexers provided in Fig. 14.14, design fancy shift register (barrel shifter) specified in Table 14.6. It should be configured as a Johnson counter for all inputs that wrap-around in either direction. It must have the following capabilities:

14.6 Design a 3-bit binary up-down mod 6 counter using D flip flops. It should count from 0 to 5 and then reset to 0. Include an input X that will count down if 0 and up if 1. Write down the state table and derive the minimal logic for D_2, D_1, and D_0. (Hint: the state table should have 4 entries: Q_2, Q_1, Q_0, and X. You don't need to worry about output for the counter.)

14.7 Design a 3-bit counter that counts down from 6 to 1.

14.8 Design a 4-bit binary counter with a control input C. When C = 1, the counter should count down one. When C = 0 the counter should maintain its current value. The counter should start at 12 and count down to 1 before wrapping around to 12 again.

14.9 Design a 3-bit mod-5 binary/Gray code counter with two inputs B and G. When B = 1 it should count up in the binary sequence and when G = 1 it should count up in the Gray code sequence. (Assume B and G will never both be 1 at the same time.) Each count sequence should roll over from its largest permissible value to 0.

14.10 Design a binary counter which counts down from 7 to 2 and then resets. Use don't cares for unused states. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

14.11 Design a four-bit counter which counts in the Gray Code (Table 9.2 in the text) sequence and resets when it reaches the end. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

14.12 Design a binary up/down mod 5 counter. If the input $c = 0$ then count down, and if $c = 1$ then count up. Use don't cares for unused states. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

14.13 Design a 4-bit twisted ring counter. A twisted ring counter follows the count sequence 0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000, and then back to 0000. Use don't cares for unused states. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

14.14 Design a binary up/down mod 6 counter. If the input $c = 0$ then count up, and if $c = 1$ then count down. Use don't cares for unused states. Fill in the truth table and calculate the next state logic. Use SR flip flops for the state memory.

14.15 Design a 3-bit up/down twisted ring counter. A twisted ring counter follows the count sequence 000, 001, 011, 111, 110, 100, and then back around to 000. It should count up when the input $c = 0$ and down when the input $c = 1$. Use JK flip flops for the state memory.

14.16 Implement a synchronous sequential circuit to output the sequence 0123401234... with a reset input (R) such that the next digit in the sequence is output when R = 1 and the digit 0 is output when R = 0. Implement this machine using D flip flops by using the truth table on this page and the K-maps on this and the following pages. Take advantage of any don't cares that come up. Finally, implement the circuit on the page following the K-maps.

14.17 Design a binary counter which counts down from 7 to 2 and then resets. Use don't cares for unused states. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

14.18 Design a binary up/down mod 5 counter. If the input $c = 0$ then count down, and if $c = 1$ then count up. Use don't cares for unused states. Fill in the truth table, draw the state diagram, and calculate the next state logic. Use DFF's for the state memory.

Chapter 15
Datapaths

Now that we've discussed a number of datapath components we're ready to combine them into actual *datapaths*. What we are doing here is implementing computations directly in digital circuitry. The datapaths we build are going to have the ability to perform a variety of computations. In succeeding chapters we'll connect another digital circuit network called a *controller* to the datapath to fully implement an algorithm. In this chapter we simply put in place the necessary machinery to support whatever computations such an algorithm would require. Not all datapaths need a controller, and we'll start with those.

Example: Guessing Game

Let's design a datapath that can implement a guessing game system. We'll assume we have a digital scale input to the datapath, call it W, that gives the weight of an object. We also have a keypad input, call it K, that provides an unsigned integer input indicating the user's guess for the weight of the object. The system should provide one of three outputs: LOW if the guess is less than 10 below the correct weight, HIGH if the guess is more than 10 above the correct weight, and CORRECT if the guess is within 10 of the correct weight. The value 10 here is a tolerance and can be adjusted easily; it's just for illustration purposes here.

In Fig. 15.1, we see the high level block diagram for the datapath.

To think about how to put the components together, let's first start with one of the outputs, say LOW. We need to convert the following programming statement into hardware:

```
if (K < W - 10)
      LOW = 1;
else
      LOW = 0;
```


J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_15

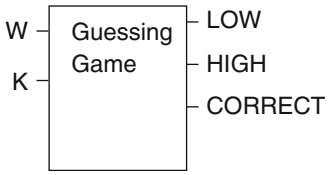


Fig. 15.1 Block diagram for guessing game datapath

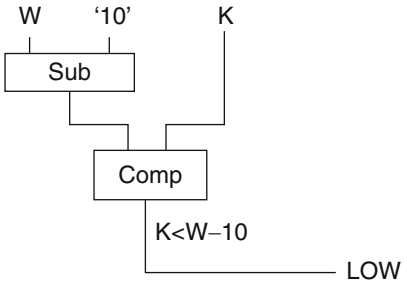


Fig. 15.2 Guessing game datapath to compute the LOW output

What we need to do is evaluate the expression `K < W - 10` in our hardware and connect that value to our LOW output. In some languages this would be considered *evaluating a Boolean expression*. In the C programming language it's just an integer 1 or 0, and that's because C is a low-ish level language that reflects the actual workings of the digital logic layer of the computer. While remembering that we don't at all unthinkingly assign the logic-1 value to the Boolean value True nor the logic-0 to False, we'll be able to compute the LOW output pretty directly here by utilizing a *subtractor* and *comparator*. This is why we have to remember what our components do and how to hook them up to input and output signals—they directly represent the basic things we can do within a datapath. Whenever we have a given computation to perform, it is the standard datapath components that we look at first in order to see how we can calculate the desired quantities. And, since we know how to specify logic functions from scratch and minimize our Boolean equations, we're always able to construct a specialty device to perform a computation when needed. We tend to favor the standard components due to cost issues, but we are designers after all and new designs are never completely ruled out.

In Fig. 15.2 you can see the first portion of the datapath that computes the LOW output signal.

As Fig. 15.3 shows, We can likewise pretty easily connect similar components to compute the HIGH output.

It's important to connect in your head the programming constructs from the code given above and the datapath components we're using here. The subtractor and adder are pretty intuitive, but we need to start connecting the comparator as the device that evaluates Boolean expressions. Computing professionals are often versed in programming to one degree or another, and it's usually pretty straightforward for us to scrawl down a pseudocode algorithm to do a thing we're interested

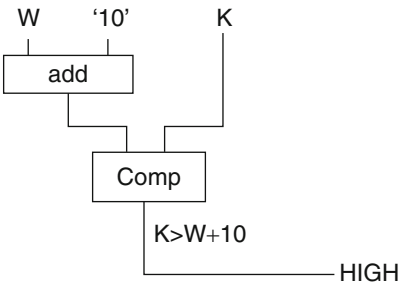


Fig. 15.3 Guessing game datapath to compute the HIGH output

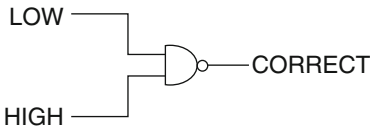


Fig. 15.4 Guessing game datapath for CORRECT output

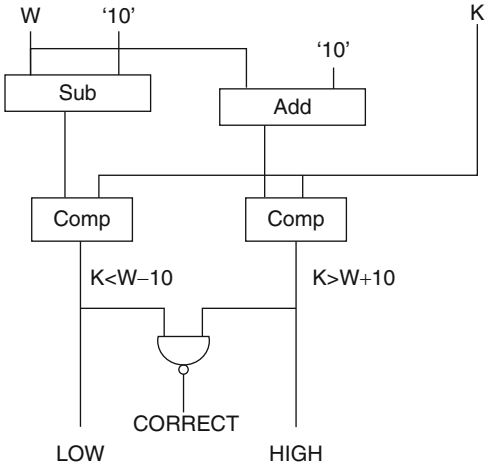


Fig. 15.5 Complete guessing game datapath

in. The thing to note here is that we can directly translate the pseudocode (or actual code in most cases) to datapath design. It's a theme we fully explore in Chap. 18 on Datapath Controllers, but it's worth beginning to think about in depth now.

The final output, CORRECT, occurs when neither of the other occur. So we can use our basic Boolean connectives to build the circuit for that output, as seen in Fig. 15.4.

Putting a NAND gate to good use, we can now see that exactly one of the three outputs LOW, HIGH, and CORRECT are going to be active at once, and that's exactly what we want.

We can put these three separate diagrams together in a final datapath that implements the algorithm in the original problem statement. Behold, Fig. 15.5.

Now, there are certainly efficiencies to be had here. The ALU components, subtractor and adder, are expensive and maybe we can combine them in some way. Maybe we can do some mathematics on the Boolean expression and cut down the comparators, etc. But, for a first pass this is great. And, there does not exist some mechanical process akin to K-maps that we can apply to datapath minimization. It's like algorithm construction itself—part craft, maybe part art—all the output of a human mind. We can let computational tools do most of the work by writing high-level behavioral programs using a *hardware description language* such as Verilog, System C, or VHDL. These tools will go straight from our C-style algorithm description to a transistor array layout. Sometimes this is desirable and a useful shortcut in the digital design process, but sometimes we want more control. The aim of this book is to help you understand the lower-level of design and then, if you become a professional digital designer and use hardware description languages every day in your work, you'll know what's going on. For our purposes, then, the above version of the datapath is exactly where we want to end up.

It's also good to start thinking in terms of these algorithm-hardware connections. We'll get to it more in later chapters, but we should extend the fact that we can take algorithms from our heads to programming languages to the next level, which is taking them all the way to the hardware. That's the key theme of this book: algorithm from abstract space to physical reality.

Example: Minimum Values

Let's try another relatively simple example to help emphasize the connection between algorithms and these datapaths we're constructing. In this case, let's take a simple computation and build the datapath. Say we have three unsigned inputs A, B, and C, and we want to compute A – min(B,C).

This requires us to work again with an selection statement, but this time the two possible statements do more than simply set a value to 1 or 0. In this case we have the following, where we'll use Z to indicate the output:

```
if (B < C)
    Z = A - B;
else
    Z = A - C;
```

Whereas previously we could just connect the output of the Boolean expression evaluating comparator directly to the appropriate system output, this time we'll need to be more careful. We are facing a situation where we have to select one of two values that are not simply logic-1 or logic-0 to attach to the output. Here we really have to convert the `if .. then` programming idiom into our hardware circuitry. To do that we need to use both the comparator component to evaluate the

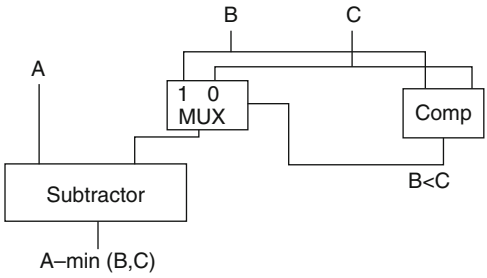


Fig. 15.6 Datapath for A – min(B,C) computation

Boolean expression as well as a multiplexor component to implement the selection part. Figure 15.6 illustrates the technique.

The output of the comparator, representing B < C, is used as the select control signal for the MUX, which allows either B or C to pass through into the subtractor.

It's so important to see this connection! We'll be using this for pretty much every selection statement found in an algorithm.

Now let's look at a more involved example.

Example: Detection Unit

Let's say we have a hand-held detection unit that we can walk around with, take sensor readings with, and press buttons to perform a few functions. We can build this thing in any number of ways. For discussion's sake, let's suppose we have three input readings A, B, and C, and four buttons B0 through B3. We have two output lights L0 and L1 and want to implement the following functionality:

B0: store A and B into registers RA and RB and clear register RC
B1: copy the value in RA into RB and store C into register RC
B2: turn on L0 if A < B and the MSB of RC is 1
B3: turn on L1 if RC < 40 and RB < 30

We'll build this step-by-step and illustrate the process as well as changes that you'll have to make as you construct your datapaths. Just like for a large algorithm you may be designing, you don't have to construct the perfect final version from scratch all at once. It's a process. Let's start with the first button's requirements.

The components required are three new registers. So, we'll start our datapath with three registers RA, RB, and RC. We know how registers work: that they have a single input line, a single output line, and a LD control signal. Registers may have other control signals to shift, set, clear, increment/decrement, etc., and we can keep augmenting our registers with extra features if we need to. In Fig. 15.7, we see the datapath that can implement the demands of B0:

Notice both what's here and what's **not** here. We have the three registers, we have the B0 input tied to the LD inputs for RA and RB, and we have the A and B inputs attached to the appropriate registers. However, we have yet to hook up the

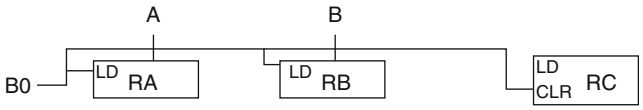


Fig. 15.7 First step of implementing the detection unit

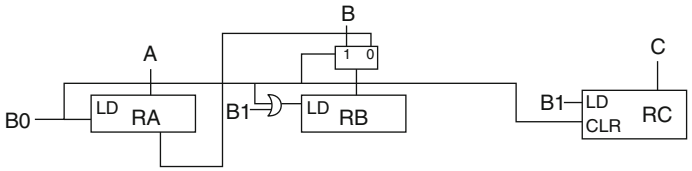


Fig. 15.8 Second step of implementing the detection unit

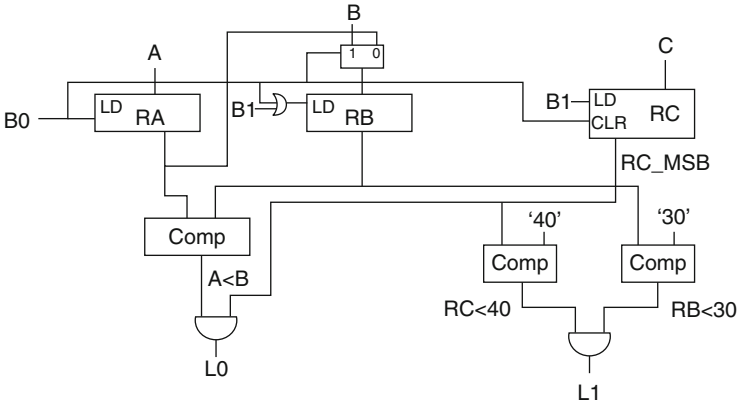


Fig. 15.9 Completed datapath design for detection unit

RC LD signal or the input to RC. That's OK! We are building this as a step-by-step process and we don't have to fill in every single detail yet.

We can now augment the datapath with the requirements of B1. This adds two things: a second input to RB and a first input to RC. We'll have to add a MUX to RB's input to account for the fact that we now have two sources of input for that register. This is actually one of the biggest changes necessitated as we go along through these sorts of design processes—you may want to always leave room for multiplexing the inputs to registers. The new datapath is diagrammed in Fig. 15.8.

A couple of changes had to be made here. We see the MUX selecting one of two inputs to RB as well as the input C heading into RC now. But, we also have to use an OR gate for the LD signal for RB because we are now updating the value of that register and so the LD signal needs to be high when either B0 or B1 is logic-1.

The next two buttons give the requirements necessary to turn out the output lights L0 and L1. Neither of these additions will require changes in the current register framework, so we can add the light signals in one step without conflicting with our previous design work. Figure 15.9 showcases the final result.

We had to add an output to register RC pulling the MSB (and only the MSB) into an AND gate with the output of the A < B comparator to implement the full condition for L0 to light. The L1 light was just a straightforward application of comparators.

You should be able to envision many permutations of this sort of design and start seeing that these digital datapath components we've built in the past few chapters are all ordered towards the implementation of particular computations. When we need to build a device that requires particular computations, we can use these components.

Algorithms

The previous datapaths are all self-contained in the sense that all necessary control signals for the components are generated either by defined inputs to the datapath or by computations conducted within the datapath. In order to implement more complex sequences of computations, we'll often have to do more. There are times when not every required control signal can be generated directly within the datapath. This is because some signals are the result of a *past sequence* of events that have occurred. In these cases, we'll need to utilize the powers of sequential logic in the form of a state machine to input these control signals into the datapath.

The clearest example of where this *memory* property of control signal generation is likely to be used is in the programming idiom known as the *loop*. Whether it's count-controlled (*for*), event-controlled (*while*), or even a hybrid such as a mid-test loop, the entire idea of executing a number of computations *multiple times until a condition is met* requires remember where we're at. More generally, an algorithm involving multiple steps to be followed in a particular order demands the hardware remember which step it's on at a given moment. Given our focus on synchronized systems and the use of clock signals the control the updating of registers, we can associate with each clock tick (in principle) a particular step of an algorithm. The full development of this idea is completed in Chap. 18 on Datapath Controllers chapter; here we give some examples of algorithms that will rely on a separate digital logic system to provide its control inputs.

As before, it's important to associated different algorithmic programming idioms with the various organizations of components represented in the following discussions.

Example: GCD Calculator

Let's look at a classic algorithm in computer science, hailing all the way back to Euclid, which computes the greatest common divisor (GCD) of two inputs:

```
input unsigned integers x and y
while ( x != y )
    if ( x > y )
       x = x - y;
    else
       y = y - x;
   assert valid
```

We can see a new thing happening here: the *while* loop. Each iteration of this loop depends on register updates: the values of x and y. This means our clock signal needs to mediate the steps of this loop. While it's not necessarily the case that every loop requires externally generated control signals in order to properly function, it is very common in digital design to partition systems into datapaths and controllers. So, in the interest of building to this idea, we'll consider this algorithm from that perspective. Therefore, we have to identify two separate categories of events taking place here: (1) four computations and (2) two control structures.

For the first case, we proceed as we have before with the construction of a datapath capable of computing the required quantities. For the second, however, we need to choose between two statements in an if as well as between either executing the loop body or exiting the loop. Our design choice will involve giving the datapath responsibility for the first category while granting to a separate control unit network the responsibility of resolving the `if` and `while` statements.

So, the development of the datapath for this algorithm requires us to identify the computations to be performed while not worrying about the resolution of certain aspects of the algorithm. This lets us leave control signals unassigned in our datapath design. Unlike the previous examples we've worked in this chapter, here we do not need to compute every single control signal because we can assume these are being generated by a separate control unit. Similarly, we can compute certain quantities that are not used internally by the datapath but which are instead send to the controller. We call these *status signals* as they provide to the control unit the *status* of the datapath.

We need to identify then, given a particular algorithm, which computations are needed to be performed by the datapath. In this example, we have four computations:

1. $x = x - y$
2. $y = y - x$
3. $x \mathrel{!}= y$
4. $x > y$

The first two are pretty straightforward to see—anytime we have an assignment statement where the right-hand side represents a basic arithmetic or logic operations, it's reasonably clear to see that this is indeed a computation our datapath ought to perform. It may be nontrivial to figure out how to complete that operation, but it is at least evident that it must be something our design takes into account. The last two computations are bit subtler. These are Boolean expressions within the

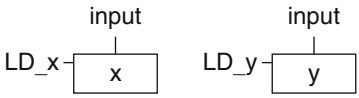


Fig. 15.10 First step of the GCD calculator datapath

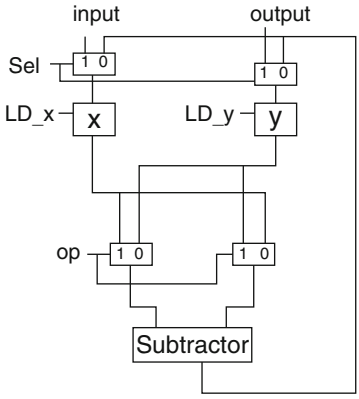


Fig. 15.11 Second step of the GCD calculator datapath

condition of if and while statements. This is where our programming mindset and instincts are so important—when writing such statements in a programming language we are well aware of the fact that these are indeed expressions which require evaluation. When seen in a pseudo-code algorithm layout we may forget this. But, they are computations indeed! And it is the job of the datapath to generate these values.

In the interest of useful pedagogy, we'll go through the design of this datapath in a step-by-step manner as we did the previous example. This will necessitate changes as we add more to it. It's recommended that the reader stop right now and try to work out the design, and then return to the text to see our development.

First, let's tackle the first step of the algorithm which inputs our numbers x and y.

In Fig. 15.10, we introduce two registers to handle storage of the variables, connect these registers to the inputs, and provide LD signals. We do not need to attach these LD signals to anything within the datapath. Indeed, we'll allow the datapath controller to produce these.

The next thing to do is to compute the two subtractions x – y and y – x. In the interest of doing a little bit of work towards efficiency, we'll use a single subtraction unit with multiplexed inputs instead of multiple ALU's. We also need to now multiplex the inputs to our registers because we can now see we have to store values from multiple sources: the subtractor as well as the input. Here's our next iteration of the datapath (Fig. 15.11):

We can see some efficiencies here in the multiplexors: we use two signals for the four MUXes. In the case of the MUXes selecting inputs for the registers, we realize that we never need to load one from the external input and one from the subtractor. So, we can use a single sel signal to differentiate between which algorithm step we're implementing. Similarly, we have a single op signal which selects between

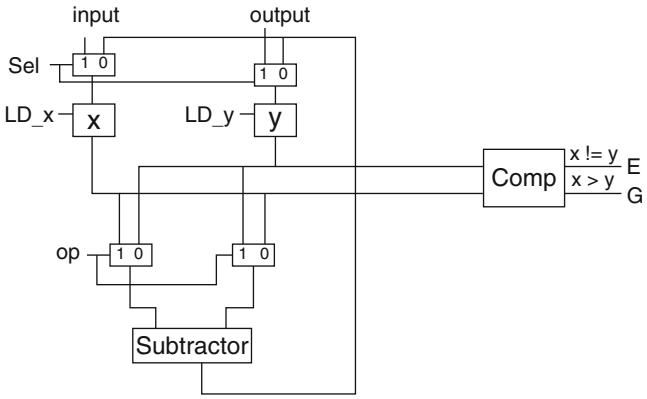


Fig. 15.12 Completed GCD calculator datapath

Table 15.1 Control word table for the GCD datapath

State	LD_x	LD_y	sel	op	valid
Input x and y	1	1	1	d	0
x = x – y	1	0	0	1	0
y = y – x	0	1	0	0	0
Assert valid	0	0	d	d	1

the two computations x – y and y – x. We don't need the other combinations x – x and y – y so we don't need two signals here. Again, as with the LD_x and LD_y signals, we leave the sel and op signals to be generated by the control unit for this datapath. The final computation we need are the Boolean expressions x! = y and x > y, so we can just add a comparator and complete the datapath (Fig. 15.12).

The comparators compute the outputs E and G which go to the control unit. We could negate the E signal to give x != y but it turns out it won't matter in the long run because the controller can handle it this way as well. One could design it either way.

The final step of the algorithm is to assert a valid signal when the computation has ended. Since this happens after the while loop has completed, and we're granting to the controller the power to determine when that takes place, we'll also allow the controller to output this valid signal. The datapath is now complete.

It's worth considering now what control signals are required for each step in the algorithm. Sometimes referred to as the *control word table*, due to the fact that the constellation of control signals sent from the controller to the datapath was originally called the *control word*, we can arrive at the framework in Table 15.1 for what the controller will be required to produce.

We can see that both LD signals need to be asserted in the *input* state while exactly one of each is asserted in the other computation states. The *op* signal is a don't care in states where we're not using the subtractor. If we're not using the output of the

subtractor, we don't need to care what inputs are being provided to it. The valid output is 1 only in the *assert valid* state, which make sense. We can also see in that state that both LD signals are 0. It's typical to always turn off the ability of registers to change in states where the variables are not being assigned. We take it for granted when writing our code that variables won't change when they are not being referenced in a statement, so in our hardware implementations we need to go ahead and make that explicit by remembering this rule. Note that we can change more than one variable at a time, as in our *input* state here, so in a way we get parallelization for free in hardware when it's really hard to get in software. But, when the algorithm specifically calls only for a single variable to change, we have to make sure we respect that in the digital design by zeroing out the LD's of unused registers.

When we design datapath controllers we'll use this sort of table as our guide to how the controller is to perform, so it's well worth figuring out how it's supposed to look now.

Example: Fibonacci Sequence

The Fibonacci sequence is one of the most studied sequences in mathematics and many computer science texts use it to study a wide variety of algorithmic techniques. So, it's well worth visiting as our second example of how to use hardware to implement algorithms. And yes, this is a huge deal—the idea that we can design specialized hardware to implement algorithms is a mind-bending thing the first time it's encountered. Programmers think everything must be funnelled through the computer itself, but specialized circuits are very much a thing and tremendously powerful. They take a lot more resources to develop, but where they can be built they outperform corresponding computers.

Recall the Fibonacci sequence begins with two 1's and progresses by adding two terms together to get the next term. So, we have 1, 1, 3, 5, 8, 13, 21, 34, etc. Mathematically, we describe it thusly:

$$F_0 = 1;$$
$$F_1 = 1;$$
$$F_n = F_{n-1} + F_{n-2};$$

We can build a datapath that reads in a number n and outputs the nth number in the Fibonacci sequence.

There are a few interesting things to be seen in Fig. 15.13. First, notice that the number n is in a counter with a decrement input that is topologically disjoint from the rest of the datapath. This is simply the count-control in the *for* loop. The condition to exit the loop is determined by the Z status signal that is sent to the control unit. Our two registers holding the Fibonacci numbers are imbued with a specialty functionality where they can both be *set* via the *init* control signal. Since we need to initialize to 1 in this algorithm we can either add multiplexors to the inputs to select from 1 or any other input, or we can embed that within the register

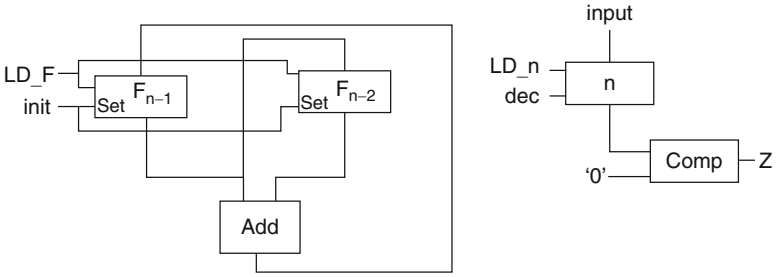


Fig. 15.13 Fibonacci sequence datapath

Table 15.2 Control word table for the Fibonacci sequence

State	LD_F	LD_n	init	dec	valid
Initialize	0	1	1	0	0
Compute $F_n = F_{n-1} + F_{n-2}$	1	0	0	1	0
Assert valid	0	0	0	0	1

as a specialized circuit. We've chosen the latter here. We are also using the same LD signal for both registers because they are both only updated at the same time, every iteration. Let this be an example that you should not feel constrained by what you've seen before. Every algorithm can sing it's own song and require a unique approach and circuit configuration.

There are only three states for the control word table here (Table 15.2):

If we inspect this carefully we can see opportunities for optimization in the control signals. LD_F and dec are the same! We could replace them with a single signal. This is interesting as it's not something we necessarily see right away when thinking about the algorithm. Rather, it's something that reveals itself when putting the design together and seeing what the control unit will be required to do. Similarly, we can combine the LD_n and init signals. When we care about minimizing the design these opportunities will be valuable. Since optimization tools usually do a better job of this, we won't often be focusing on this type of reduction. Still, it's good practice to take the opportunities when they are pretty clear.

Exercises

15.1 Consider the following instructions/operations:

MOV B, A
MOV C, B
MOV C, A
MOV B, C

Draw a datapath with three registers A, B, and C which can execute the four instructions in the sidebar to the left. Use registers and MUXes in your design. Note that you only need to execute these instructions; you don't need to include the Fetch/Decode subsystem.

15.2 Design a device that will take in three unsigned numbers A, B, and C. It should output the sum of the larger of A and B and the smaller of B and C. That is, the output $= \max(A, B) + \min(B, C)$. You may use MUXes, comparators, and a single adder in your design.

15.3 Build a guessing game system. The player has to guess the weight of an object. If the guess is within 10 ounces, then a signal is output to turn on an active-low light which indicates the player has won. If the guess is not within 10 ounces, then the light remains off.

15.4 The inputs to the system are the weight W (in ounces) of the object read from a digital scale and the guess G of the player read from a keypad. You may use MUXes, comparators, adder/subtractors, and logic gates in your design.

15.5 Design a datapath which inputs three 3-bit unsigned numbers x, y, and z. If z is the binary representation for 3 or 7, output x + y. If z is the binary representation of 4 or 6, output x + z. If z is anything else, output y + z. You may use comparators, adders, decoders, and multiplexers in your design.

15.6 Design a datapath which inputs three 8-bit unsigned numbers x, y, and z, and outputs $x - y$ if $x - y > z$ and outputs $x + y$ otherwise. You may use adder/subtractors, comparators, decoders, and multiplexers in your design.

15.7 Design a datapath which will add the value in register A to the value in either of registers B or C and store the value back in A. The input to the datapath should be a control signal s: if $s = 1$ then A + B should be calculated; if $s = 0$ then it should be A + C. You may use any of our basic datapath components in your design.

15.8 Design the electronics for a hand-held sensing device which can input two readings A and B, blink a light or sound a buzzer based on these readings, and transmit recorded readings serially to another device.

The inputs to the datapath are sensor readings A and B, an active-high reset button R, an active-high sense button S, and an active-low transmit button T.

When reset is pressed, the device should clear two internal counters X and Y.

When the sense button is pressed:

(1) X should be incremented if the reading A is greater than 20
(2) Y should be incremented if the reading B is less than the reading A or if the reading B is greater than 10
(3) both readings A and B should be stored in internal shift registers

An active-low light (L) should be turned on if counter X has been incremented at least 10 times.

An active-high buzzer (Z) should sound if the counter Y has been incremented at least 15 times.

When the transmit button is pressed, the MSB of the stored values of A and B should be output and the values should be shifted one to the left.

You may use any of our digital components in your design. Be sure to label all your inputs and outputs.

15.9 Design a datapath which will add the value in register A to the value in any of registers R0 through R3 and store the result back into any of registers R0 through R3. The inputs to the datapath should be two-bit control signals *src* and *dest*. The signal *src* will choose among the four registers to be added and *dest* will choose among the registers in which the result is to be stored. You may use any of the basic datapath elements in your design.

15.10 Design a datapath which will do one of the following: (1) add the value in register A to the value in one of the four registers R0 through R3 and store the result back in A, (2) subtract the value in one of the four registers R0 through R3 from the value in register A and store the result back in A, or (3) move the value from one of the four registers R0 through R3 into register A. The inputs to the datapath should be control signals *op*, *sel*, and *in*. The signal *op* is 1-bit: if *op* is 1 then add, if *op* is 0 then subtract. The signal *sel* is two bits and chooses which of the four registers will be involved in the operation. The signal *in* is one bit: if $in = 1$ then perform addition or subtraction, if $in = 0$ then perform the move operation. You may use any of the basic datapath elements in your design.

15.11 Design a datapath which has registers R0 through R3 and will output the value R0 + R1 if the value in register Z is zero and output R2 – R3 if the value in register Z is nonzero. You may use any of our basic datapath elements in your design but use only a single adder/subtractor unit.

Design a datapath which can implement any of the following operations:

- Load a value from the memory address stored in PC into the register IR
- Increment the value in the PC register by 1
- Add the value in register A to the value in any of the registers R0 through R3 and store the result in A
- Store the value in register A into the memory location at the address given by register MA

Clearly label your control signals. You may use any of the basic datapath elements in your design. Also fill out a table indicating what signals should be asserted in order to accomplish each operation.

15.12 Design the electronics for a hand-held sensing device which can input two readings A and B, blink a light or sound a buzzer based on these readings, and transmit recorded readings serially to another device.

The inputs to the datapath are sensor readings A and B, an active-high reset button R, an active-high sense button S, and an active-low transmit button T.

When reset is pressed, the device should clear two internal counters X and Y.

When the sense button is pressed:

(1) X should be incremented if the reading A is greater than 20
(2) Y should be incremented if the reading B is less than the reading A or if the reading B is greater than 10
(3) both readings A and B should be stored in internal shift registers

An active-low light (L) should be turned on if counter X has been incremented at least 10 times. An active-high buzzer (Z) should sound if the counter Y has been incremented at least 15 times.

When the transmit button is pressed, the MSB of the stored values of A and B should be output and the values should be shifted one to the left.

You may use any of our digital components in your design, including shift registers and counters. Do not use a system clock; instead assume the registers update only based on the value of the control signal inputs. Be sure to label all your inputs and outputs.

15.13 Design a datapath which can implement any of the following operations:

- Load a value from the address stored in register MA into any of the registers R0 through R3
- Add or subtract the value in any of the registers R0 through R3 from the value in register A
- Move the value from A to any of the registers R0 through R3 or to the register MD
- Store the value from MD into the memory location given by the address in MA

Clearly label your control signals. You may use any of the basic datapath elements in your design. Also explain what signals are required to be asserted in order to execute each of the given operations.

15.14 Design a datapath which can implement the following algorithm:

```
input x, y, and z
while (z > 0){
  if x > 5, x = x - y
    else y = y + x
  z = z - 1
}
output x and y
```

Clearly label your control signals and status signals and indicate which signals must be asserted for each computation required of the datapath.

15.15 Design a datapath which can sum inputs until the inputted value is 0 at which point the datapath will output the sum. You may use any of our basic datapath elements. (Do not worry about the issue of possibly overflowing the sum register; we'll just assume it's large enough for what we need.) Write the algorithm, diagram the datapath, and clearly label all control and status signals. Also, indicate which signals are to be asserted for each computation required of the datapath.

15.16 Design a datapath which can implement any of the following operations:

- Load a value from the memory address stored in PC into the register IR or the register MA
- Load a value from the memory address stored in MA into the register A
- Add the value in register A to the value in any of the registers R0 through R7 and store the result in A
- Move the value from register A into register PC if the value in IR is equal to 15.

Clearly label your control signals and status signal. You may use any of our basic datapath elements in your design as well as a memory unit.

15.17 Consider the datapath given in Fig. 15.14.

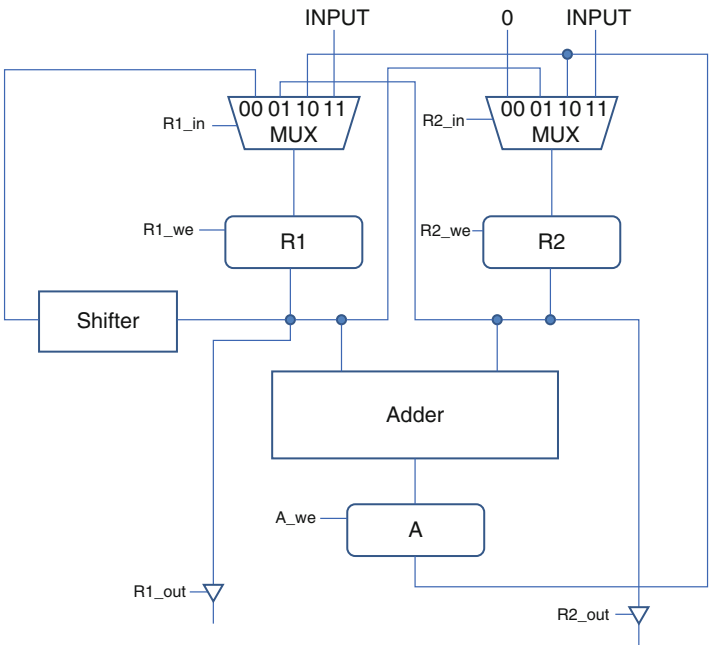


Fig. 15.14 Datapath for exercise 15.17

(a) Determine the logic levels for each control signal necessary for this datapath to execute each of the following instructions. Take advantage of any don't cares that you can. Note that the signals R1_in and R2_in are each two bits.

Instruction	
LOAD R1	Load value from INPUT into R1
STORE R2	Send value from R2 out
ADD R1, R2	Update A with the value R1 + R2
MOV R1, A	Update R1 with the value from A
MOV R2, R1	Update R2 with the value from R1
SHR R1	Update R1 with R1 shifted right
CLR R2	Set the value of R2 to 0

(b) Explain why this datapath **cannot** execute the instruction MOV A, R1 which would update the register A with the value from R1.
(c) Write a two-statement program using the instructions given which will update the register A with the value from R1.
(d) Show how you would modify the datapath to allow execution of the MOV A, R1 instruction directly. Draw your modifications on the datapath diagram.

15.18 Design a datapath that will take in three numbers A, B, and C and output A + min(B,C) + max (A,C).

15.19 We want to build a device that can store two sensor readings and compare another reading to them. The inputs are a reading R and active-high buttons B0-B3. The outputs are active-high lights L1 and L2.

The buttons should perform the following functions:

B0 store the input R in a register A
B1 store the input R in a register B
B2 clear both registers A and B (that is, set them equal to 0)
B3: do both of the following:

(1) turn on L1 if the stored reading in A is greater than the input R
(2) turn on L2 if the stored reading in B is no more than 5 less than the input R

You may use any of our digital components in your design. Be sure to label all your **inputs** (B0-B3, R) as well as your **outputs** (L1 and L2).

15.20 We want to design the electronics inside a stuffed toy bear that responds to a child's actions.

The device has the following sensor inputs: voice **V**, motion **M**, acceleration **A**, and touch **T**.

It also has a calibrate button **C** as well as a mute switch **U**.

The outputs cause its eyes to flash in sync with a tune (**E**) and cause the bear to laugh (**L**).

When the calibrate button is pressed (**C = 1**), the current values of V and M are stored in internal registers.

When a parent foolishly unmutes the toy (**U = 0**), the device will respond to the child in the following ways:

(1) if the input V is greater than the stored level of V, or if the input T is greater than 5, then the eyes flash (E = 1)
(2) if the sum of the inputs M and A is more than 10 greater than the stored value of M, then the bear laughs (L = 1)

You may use any of our digital components in your design. Be sure to label the **inputs** and **outputs**.

15.21 Consider a system which reads in N numbers and then outputs the maximum, minimum, and average values of those numbers.

The system should wait for a start signal S, input the value N, then input the numbers one by one and perform the necessary calculations. Finally, when the N^{th} number has been input, a valid bit should be set and the three required quantities output.

You may assume the input numbers are available whenever you need them; there is no need to wait on an enter signal to indicate they are ready.

You may use any of the digital components, including an ALU capable of adding and dividing.

15.22 Design a device that contains three internal registers A, B, and C, and inputs a two bit control signal s_1s_0. The system should perform the following operations based on the input:

s_1s_0	
0 0	Add A + B and store the result in C
0 1	Add B + C and store the result in A
1 0	Copy the value stored in A into C
1 1	Clear both A and B and calculate the two's complement of C (storing the result back in C)

In addition to the three registers A, B, and C, you may use any number of multiplexers, decoders, and logic gates in your design. You may also use **a single adder** in the design.

Chapter 16
Basic Computer Datapath

This is the chapter you've been waiting for, what we've been building towards. We can finally construct a major part of a computer! We started with the instructions covered in the Encoding Code chapter and proceeded to learn about datapath components such as registers, MUX's, and decoders. Last chapter discussed datapaths for generic algorithms. Now we can begin to put those components together into a system capable of processing instructions.

At first blush, this may seem a daunting task: how can we design hardware to run our programs if we get to write the programs ourselves? It's not like the computer knows what program we are going to write, right? What we need to do is figure out a *meta-algorithm*: a procedure capable of executing any algorithm we may put together. Then, the digital system we design just has to put this meta-algorithm into motion and the computer should compute.

This is just what we do. We call this meta-algorithm the **Instruction Cycle.**

The Instruction Cycle

Recall that in the last chapter we focused on implementing a specific algorithm with our datapaths. For a computer we want to implement an algorithm—the Instruction Cycle—that runs our programs. So, think of this as a specialized datapath design focusing on a single algorithm (or family of algorithms.) Since this is the computer itself, it's rather important and is worthy of its own focus. This chapter just introduces the basics of the field. If you're interested in how other algorithms are utilized in the design of computers or how to further optimize the datapaths presented in this section, then the study of Computer Architecture and Organization is for you! Go get the latest book by Patterson and Hennessey and enjoy!

OK, so what is this Instruction Cycle? You may recall the beginning of the discussion in a previous chapter. We'll review that here.

The first step is to recall our original computer system diagram (Fig. 16.1).

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_16

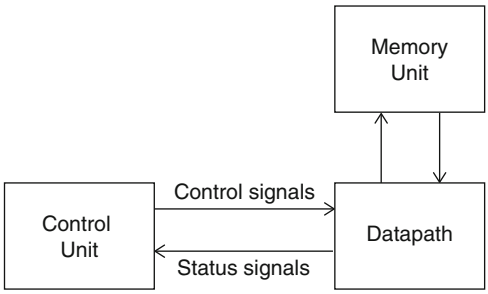


Fig. 16.1 Computer system block diagram

What does the datapath component need to do in order to process our instructions? We want our computer to **sequentially execute instructions**. Our instructions are stored in memory and must be retrieved one by one, in order, and pushed through the datapath. Computer memory is organized into locations which hold data. Each data items has a unique location identifier called an *address*. So, to retrieve anything from memory, be it a data item or an instruction to be processed, requires knowing the address of the thing and sending that to the memory unit. We then need store the value retrieved from the addressed location.

To help us do this we have two special registers. One, the **program counter**, or PC, stores the location of the next instruction in the sequence to be executed. The other is the **Instruction Register, IR,** which stores the actual 1's and 0's encoding of the instruction after it's brought into the computer datapath from its residence in memory. Together the PC and IR let us control the operation of our computing machine.

Our three-step algorithm, then, which we call the **Instruction Cycle,** is as follows:

1. Retrieve the instruction encoding from memory
2. Get the operands ready
3. Execute the Instruction

We call these three stages of the instruction cycle **Fetch, Decode**, and **Execute**.

During instruction **Fetch**, the value in the PC is sent to memory and the value in the corresponding memory location is sent back and stored in the IR. We then increment the PC so it is ready to go fetch the next instruction in the sequence.

During the **Decode** stage, all the myriad ways we have of encoding operands are unpacked and we go find the operand and typically load it in some sort of buffer register to await passage through the appropriate functional unit indicated by the operation. More sophisticated addressing modes can require longer or otherwise more expensive decode stages in a computer. This is one reason to be careful regarding the design of such encoding modes and yet another tradeoff to keep in mind when putting together a computer system.

The **Execute** stage works as expected: the instruction is executed by sending the signals through the adder or multiplier or XOR unit or whatever computational

component is required and the results stored back in the appropriate destination register.

The simple datapaths we initially consider are what we call **single-cycle machines** in that they inherently rush through all three stages very quickly—within a single clock cycle. Later we'll look at more involved designs that require the full three stages of the instruction cycle.

It's important to remember that all a computer does is fetch instruction from memory, decode operands, and execute them. Rinse, repeat, rinse, repeat ad infinitum. It's all high and low voltages running through wires repeating that cycle over and over and over. It changes your relationship with a computer when you really, deep down, get that. Think about it. Take it with you on a long walk or run or swim. It'll be there for you later, as we'll keep building on this throughout the remainder of the text.

The Fetch Stage

We'll first look at the Fetch stage of the Instruction Cycle and see what it demands of the datapath. Because this is the stage that goes out and gets the instruction for processing, it must by its very nature be independent of the exact instruction. Whereas the Decode and Execute stages will vary based on the instruction, the Fetch stage is the same for all instructions. It utilizes the PC and IR, as described above. In Fig. 16.2, we see the general datapath layout for the Fetch stage.

We see the two registers with, as we'd expect from our work last chapter, the requisite LD signals. The PC, being a counter and all, also has an increment signal. The value of the PC is sent out of the datapath to the memory unit while the input to the IR comes from the same memory unit.

From computer to computer, there can be a lot of variation even on this simple theme. A system with instructions occupying more than a single memory location, for example, may have a PC increment of +4 or +8 instead of +1, for example. One common difference you'll see often is the introduction of buffer registers for both

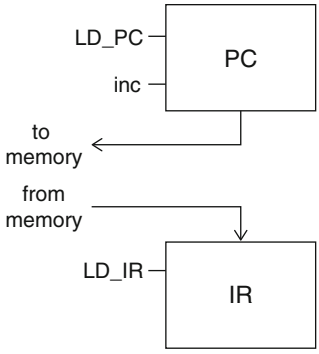


Fig. 16.2 Fetch stage datapath

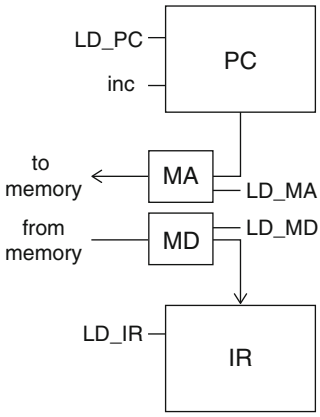


Fig. 16.3 Fetch stage augmented with MA and MD registers

the PC and IR. Often called the Memory Address (MA) and Memory Data (MD) registers, they may appear as seen in Fig. 16.3.

The reason for using these additional registers has nothing to do with the logic of the algorithm but with the physical technical requirements of the system components. Since our focus here is more on the computing logic design of the system rather than the electrical engineering level, we'll tend away from the use of these buffer registers in our designs. But, you'll often see them in datapaths and they are worth keeping in mind. When you enter into computations about clock speed and register setup and hold times, it becomes necessary to consider buffers in a variety of locations throughout a digital system, not just in this specific instance.

To sum up, here are the operations necessary to the implementation of the Fetch stage of the Instruction Cycle:

F0: mem_addr = PC
F1: PC = PC + 1
F2: IR = mem_data

As we have done before, we could specify the *control words* necessary to implement each of these operations. We'll do this in the chapter on Control Units for instruction processors. These steps are sometimes called *micro-operations* and computers can be discussed at the level of these micro-operations which drive all algorithm performance. These will need to be augmented if we include the MD and MA buffer registers, of course.

The Decode Stage

Things get much more complicated in the Decode stage. This stage needs to retrieve the operands for each instruction and place them in appropriate buffer registers. Since there are so many ways to encode operands, called *addressing modes*, there

are many configurations the Decode stage can take. We'll review some of the most commonly used addressing modes in this section and detail the resulting datapath requirements. This discussion will go into more depth than we saw in the Encoding Code chapter because we now have the full suite of datapath components available to us and experience working with datapaths in general.

Register

The most straightforward addressing mode as far as the datapath is concerned is register addressing. In this addressing mode the operand required by the instruction is located in a general purpose register located within the datapath itself. This requires a register file containing the general purpose registers as well as the ALU and its buffer registers. The basic setup is seen in Fig. 16.4.

We can see on the left of this diagram the basic Fetch machinery. We don't really need the PC or IR for this addressing mode, but we'll include these components for completeness. The ALU we can see is buffered by X and Y registers, each with their own LD signals as we've come to expect for registers. Finally, we have the register file Rn with its two output channels A and B. The sel_A and sel_B lines select which registers are output on these channels. The sel_w input will select which register in the file is to be updated, or written to, and the LD_R signal turns on and off the load functionality. We often speak of a register being loaded as the register being *written to* and sometimes we see something like WE, for write enable, as the LD signal for a register file. However it's depicted, the concept is the same: this component can output two registers and update a register all at the same time within the same clock cycle. These three operations can even be all performed on the same register! This is a powerful component and enables likewise powerful instructions. We will denote the registers in specific as R0, R1, R2, etc. When referring to them in general we'll subscript with letters such as Ra, Rb, Rc, etc. This follows the standard way to look at them common among instruction sets in use today.

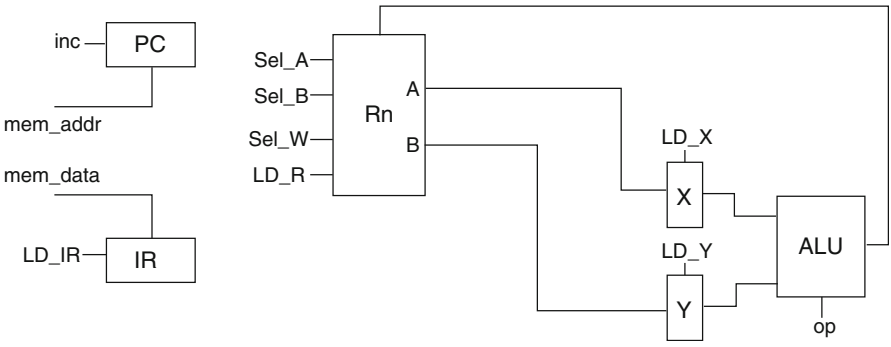


Fig. 16.4 Datapath for register addressing mode

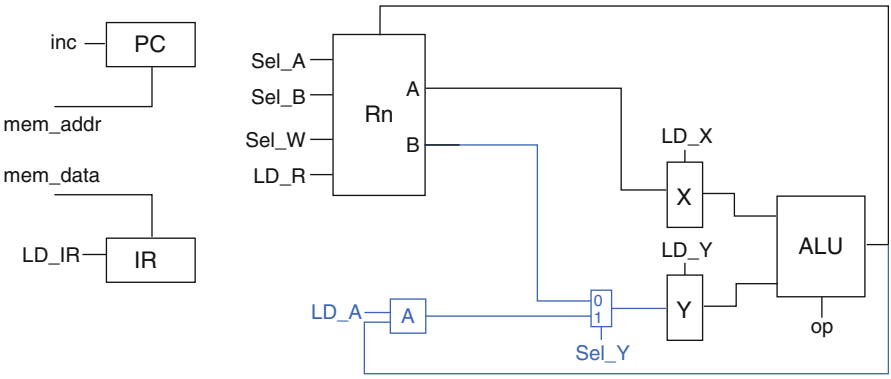


Fig. 16.5 Accumulator-based datapath for register addressing

This datapath framework can support instructions such as `alu ra, rb, rc`, where we perform some ALU operation (denoted the *op* control signal) on `rb` and `rc` and store the result back in `ra`. We can also perform a `mov ra, rb` operation where we copy the value from `rb` into `ra`. Typically the ALU has a pass_through function where it can take one of the inputs X or Y and simply output it unchanged. The reasons for running an operand through the ALU even though we're not modifying it is imply efficiency. It turns out it's more efficient to always go through the ALU rather than having special connections among all sorts of components. It also comes into play when we apply optimization techniques such a *pipelining* to the datapath, and this is something all modern day computers use, so it's good practice to get into the habit of looking at the datapath in this way from the beginning of your study.

We can even look at specialty datapaths that have registers beyond the general purpose register file. If we have an accumulator A or some other register, we can simply add that to the datapath and multiplex the buffer registers X and Y accordingly to allow for the loading of A to either. For example, if we have the instruction ADD A, Ra which will add A and Ra and then store the result back in A we need to add to our diagram (1) a register A, (2) a MUX for the ALU buffer X which permits a connection from A, and (3) a connection from the ALU back to the register A. The diagram would appear as in Fig. 16.5.

One can see how you can keep adding more and more paths and special function registers to support more instructions. As long as the operand is located in the register file, or in another datapath register added near the register file, we consider it to fall within the realm of this addressing mode.

Immediate Addressing

While the Instruction Register IR is a register in the datapath, since it's a system register and not one accessible by the programmer, its use is cordoned off and when we need to work with it it's considered to be a special case. The **immediate**

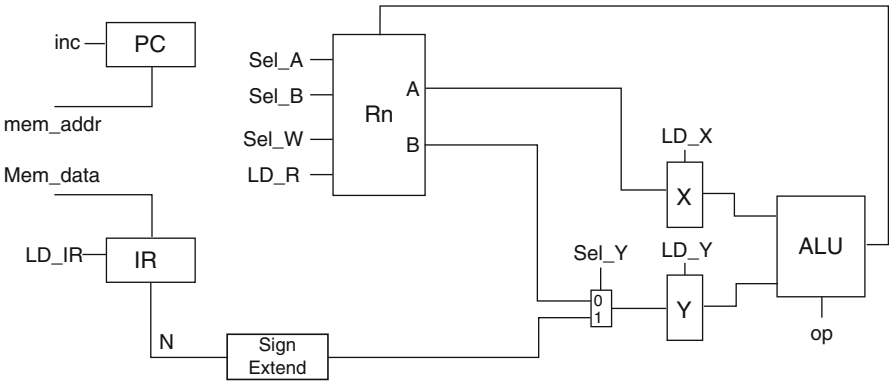


Fig. 16.6 Immediate-addressed datapath with sign-extension unit

addressing mode refers to the case where the operand we need is physically located within the IR. As we saw in the Encoding Code chapter, the restriction on the size of the immediate field necessitates *sign extension* when using this operand. Since we always need to have operands that are equal in bit width, extraction from the IR of the immediate operand requires us to also use a sign extension component that adds bits, either 1's or 0's depending on the most significant bit of the operand, to the value until it's equal to the size in other registers. Keep in mind that all our discussions here leave the width of the data bus unspecified—these are high level considerations true across a wide range of bit sizes. The only thing we are saying here is that whatever bit width we choose we must make sure the immediate operand is also of that width.

Figure 16.6 shows a datapath

We can see here that the immediate operand N is brought out of the IR (and note that we specifically label that it's the N field from the IR and not the entire contents of the IR that is being retrieved here), through the sign extender, and into the MUX for one of the ALU buffers. We can now implement instructions such as `add ra, rb, #N` or `mov ra, #N`.

Care needs to be taken to ensure the immediate operand is available in the correct ALU input channel. The above diagram shows it in the Y buffer, but that's arbitrary. We could easily have the N go into the MUX for the X buffer. It doesn't matter as long as the ALU operation under consideration is commutative. Once we have something like `sub ra, rb, #N` which requires `rb – N` to be computed, then it's imperative we use the correct input channel to the ALU.

Direct Addressing

Operands can be located in memory as well as within a datapath register or the IR. It's tough to find an operand in memory, though, because memory addresses are typically very wide numbers. They take a lot of bits to represent. So we're in a

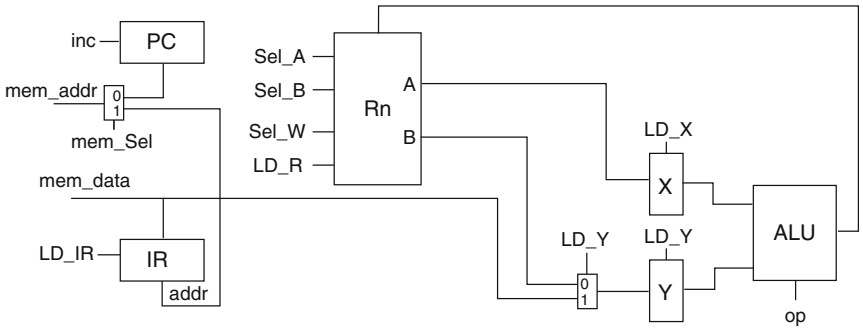


Fig. 16.7 Direct addressing mode datapath

situation where we have to figure out clever ways to compute the address. Much of this explanation is covered in the Encoding Code chapter, and here we'll follow up with the appropriate datapath diagrams. Direct addressing mode is theoretically possible but is one that has fallen out of favor beyond 8-bit variable-width architectures.

Any of the addressing modes that access operands from the memory unit will have to multiplex the *mem_addr* line. This is represented in the datapath of Fig. 16.7.

We're calling the direct address field *addr* and we use *mem_sel* to select which register holds the memory address when interrogating memory. We can pull that address either from the PC or from the IR in the case of direct addressing. As in the immediate addressing example, the diagram shows us bringing the operand into the Y buffer for the ALU, but that's arbitrary. It could just as easily be the X buffer, and in the case of certain operations such as sub it really does matter and you'll have to specialize your datapath accordingly based on the instruction set under consideration.

A sign-extend unit may also be used with the direct address field being pulled from the IR, so if that's the case for a given instruction set it will need to be added in as it is for the immediate operands.

Indirect Addressing

The concept of indirection is very important in all areas of computer science. The indirect addressing mode's motivation and use is discussed in the Encoding Code chapter. Here we'll look at the datapath requirements for indirect addressing.

Recall that in this addressing mode we store the address of the operand in memory, and the value in the addr field of the IR refers not to the operand itself but instead to the address of the operand. This means we have to go to memory **twice** during decoding of this operand. The first memory access gets the memory address of the operand and then the second memory access gets the operand itself.

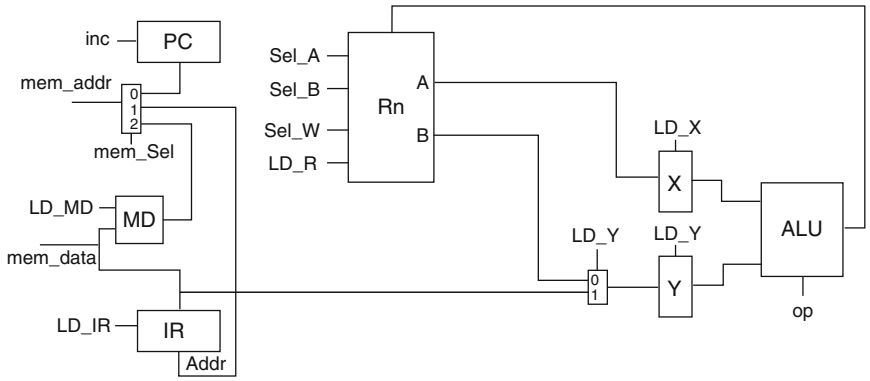


Fig. 16.8 Indirect addressing mode datapath

We are forced to use a buffer register and to consume more clock cycles for this operation because interrogating memory is a time-consuming process and for the sake of the rest of the system we don't want to slow down the synchronization signal to the point where this would be a single-cycle operation.

Figure 16.8 presents a datapath to support this addressing mode:

The sequence is as follows: (1) send addr from IR to memory, (2) store resulting value in MD, (3) send MD value back out mem_addr line, and, finally, (4) take the mem_data value and store in Y.

Again, variations exist. You can store the result elsewhere other than Y. The MD register can be used in different ways. But the core is consistent: two memory access are required and this addressing mode takes more work to decode. This should be a takeaway for the scientist of computing: more advanced algorithmic language requires more technical and temporal resources to process physically.

Indirect address structured in this way still retains the basic issue seen in direct addressing: the address itself is often too large to store in an addr field within the instruction encoding, even with the possible utilization of a sign-extension unit. In modern processors, indirect addressing often takes the form of **register indirect addressing** where the address of the address of the operand (yes, slow down and unpack that statement) is stored in a general purpose register instead of in the IR. In this case we just need to update the datapath diagram with the proper source of the address.

In Fig. 16.9 we're using the A output channel from the register file, but in principle we could use either A or B output channels. Most every modern instruction set uses a version of register indirect addressing, and a form of it is used for implementing structured data at the machine level. In particularly, a variant called **displacement addressing** is key, and that is our next topic.

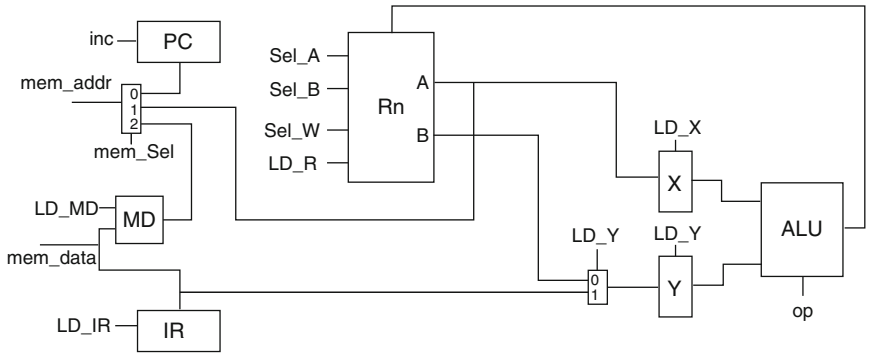


Fig. 16.9 Register indirect addressing mode datapath

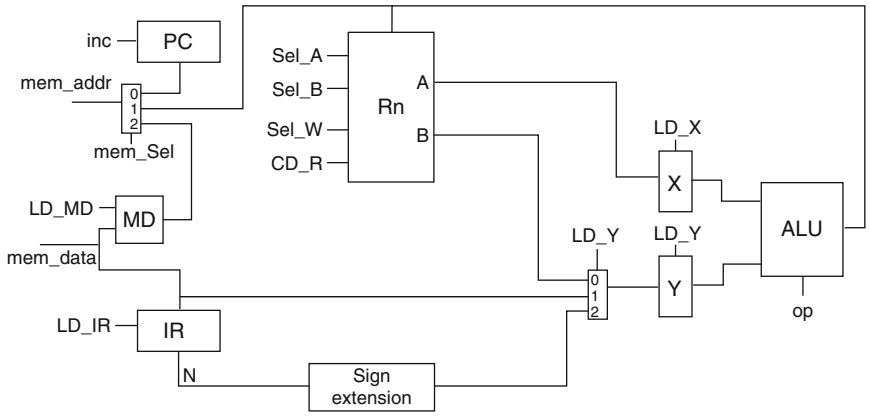


Fig. 16.10 Displacement addressing mode datapath

Displacement Addressing

We can perform operations on the addresses in registers. By leveraging the connections already in place for the register addressing instructions, we can route the address in the register through the ALU before sending it back out to the mem_addr line. This process allows us to compute an address by adding an offset to the address stored in the register. This lets us execute instructions such as `mov ra, rb(N)`, which loads into `ra` the value stored at memory location `rb + N`. We've seen previously that this can be used to access array elements, a common programming idiom.

To implement this at the datapath level, we can make connections such as those seen in Fig. 16.10.

Here is how this datapath processes the displacement addressed instruction: (1) send the register through the X channel of the ALU and the value N through the Y channel, (2) the sum is then sent through the 1 channel of the mem_addr MUX, (3) the resulting value is stored in MD before (4) being sent back to the mem_addr line to retrieve the actual operand which enters the processor along the mem_data input and gets stored in the Y buffer for the ALU. Wow, yes, read it again, trace it through the datapath, and make sure you can retell this story and really get how this critically important addressing mode is implemented in hardware.

The Execute Stage

While in principle it may be reasonable to expect that each instruction requires its own unique Execute datapath, in practice we can break all instructions into two basic categories. The first category is the ALU instructions that perform a computation via the ALU and then store the result either in a general-purpose register or back in memory. These instructions simply process the work done by the Decode stage in priming the ALU buffer registers. In fact, there is nothing else to add to the datapath diagrams we've seen already if we want to implement an ALU instruction that stores its result back in the register file! To store the result back in memory, however, requires some new datapath connections. We need to send a new address to memory, one that specifies the location to write, and we also need to have a new output channel to_mem in order to carry the data. A basic datapath to support this can be seen in Fig. 16.11.

This is based on the basic register-addressing datapath, but these elements can certainly be added to any of the more complex Decode stages. We see the mem_addr line multiplexed with the PC to support the write address (which itself may need to pass through a sign-extension unit depending on the instruction format)

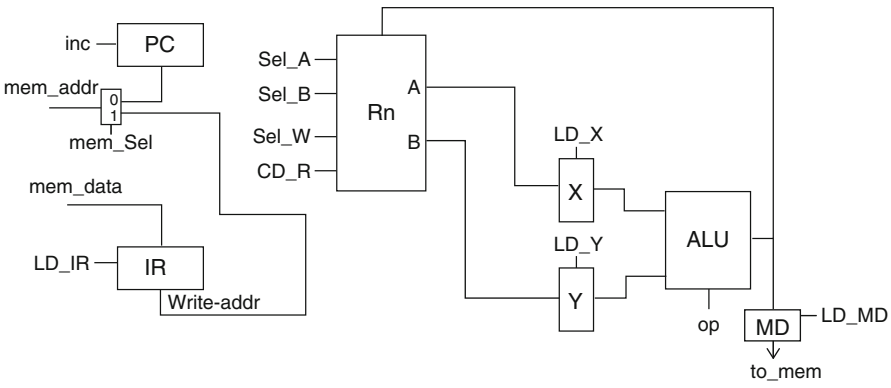


Fig. 16.11 Datapath with execute stage storing result to memory

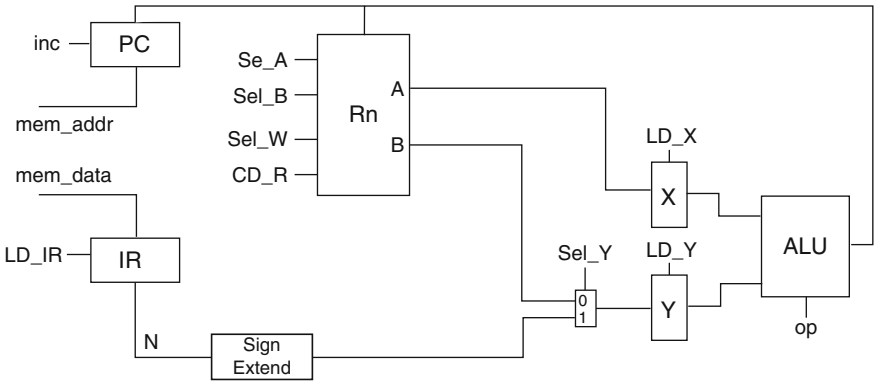


Fig. 16.12 Executing branch instructions

as well as a memory buffer MD register containing the output of the ALU. This is placed in the datapath because in general the ALU unit takes enough time to process an operation that we can't run signals through it as well as the memory unit in a given clock cycle. So, the MD buffer register is required in this case.

The other major category of instruction to be handled by the execute stage is that of branch instructions. These, covered in the Encoding Code chapter, operate by changing the sequential execution nature of the Instruction Cycle. They must override the hardware's natural incrementing of the PC to locate the next instruction in the address directly following that of the current instruction. To do this, the register that must be changed is not in the register file but is the PC itself. Therefore, we need a connection from the output of the ALU to the PC. Finally, we're using the LD_PC signal!

Figure 16.12 shows an example datapath of this type.

This datapath supports instructions such as `jump ra(N)` which computes `ra + N` and stores the result in the PC. We can construct many ways to compute the target of the branch instruction and implement them in a datapath of this kind. Maybe we add two registers together, or even retrieve a value from memory to act as the target. By combining some elements from the previous datapaths with the basic idea that the output of the ALU deposits into the PC, we can implement the branch instructions from any instruction set.

Often we need to work with *conditional branches* that update the PC only when a certain condition is met and not just automatically every time the instruction was processed. It's interesting here, now that we're looking at the hardware involved with making these instructions physically executable, to realize exactly why we have the software limitations that we do. We don't, at the hardware level, allow any Boolean expression for the evaluation of the condition of the branch instruction, whereas, as the high-level language level, we would. The reason is evident: just

look at the datapath and try to conjure up the necessary connections in order to evaluate a complex expression such as `(R2 || R3) && (!R4 ^ R5)`. We can clearly see that this quickly gets out of hand. This is why branch instructions in machine languages generally key off a specific register or are used in tandem with a test or compare instruction that generates a host of status flags for all possible Booleans expressions. The easy and workable approach is the add a single comparator that gives possible comparisons of a given register to some fixed value, such as zero. A great many instruction sets are built thusly: the conditional branch instruction is of the form `JZ target` where the mnemonic stands for "Jump if Zero".

In the datapath, all we are required to do is provide the appropriate status signal for these conditional branches. It will be the control unit's job to discern whether the PC ought to update at any given moment.

And that's it! That's a computer datapath! There are a lot of differences between the datapaths shown in this chapter and that found within your laptop or tablet, but, really, *not as many as you may think!* The study of the field of Computer Architecture will take all these ideas and run with them further. A full delve into this area is beyond the scope of this book, but this chapter is designed to serve as an introduction and teaser of sorts for the deeper results achievable by today's technologies for the production of instruction processors.

Exercises

16.1 Design a datapath that can Fetch, Decode, and Execute the following instructions.

`MOV X, #N`	Store the value N in register X
`MOV Y, Rn`	Copy the value from Rn into register Y
`MOV Rn, X`	Copy the value from register X into Rn
`ADD Rn, X, Y`	Add X + Y and store the sum in Rn
`SUB X, #N`	Subtract X – N and store the result in X
`XOR X, Y`	Compute the bitwise XOR of X and Y and store the result in X

Label all your control signals, use the values from the instruction encoding when possible, and do not include unnecessary connections. You may use all of our digital components, including a register file and a single ALU.

16.2 Design a datapath that can Fetch, Decode, and Execute the instructions given below. The `JMP` instructions copy the appropriate value (either N or A) into the PC.

Instruction	*Opcode*
MOV A, #N	**000**
MOV Rn, #N	**001**
MOV A, Rn	**010**
MOV Rn, A	**011**
ADD A, #N	**100**
ADD A, Rn	**101**
JMP N	**110**
JMP A	**111**

16.3 Design a datapath that can fetch, decode, and execute the following instructions:

MOV A, #N	loads a value N into A
MOV B, A	copies the contents of A into B
MOV C, B	copies the contents of B into C
MOV A, C	copies the contents of C into A
ADD A, B	adds A to B and stores the sum in A
SUB C, B	subtracts B from C and stores the result in C

16.4 Design a datapath that can fetch, decode, and execute the following instructions:

MOV A, #N	loads a value N into A
MOV Rn, A	copies the contents of A into Rn
MOV A, Rn	copies the contents of Rn into A
ADD A, Rn	adds A to Rn and stores the sum in A
ADD A, #N	adds A to N and stores the sum in A
XOR A, Rn	calculates the bitwise XOR of A and Rn and stores the result in A
DECNZ A	decrement the value in A if A is not 0

16.5 Use the instruction set given in exercise 16.4 to write a program which will subtract the value in R5 from the value in R2 and store the result back in R5. Note that this instruction set doesn't have a subtract instruction. You will have to figure out how to use the existing instructions to perform two's complement subtraction. Assume all numbers are 8-bits and be sure to jump for joy at the existence of the ADD A, #N instruction.

16.6 This problem asks you to both design and use a computer datapath. Consider the following instruction set.

MOV A, #N	**ADD A, Rn**
MOV Rn, #N	**ADD A, #N**
MOV A, Rn	**SUB A, Rn**
MOV Rn, A	**SUB A, #N**
XOR A, #N	**XOR A, Rn**
SHR A	**SHL A**

The SHR and SHL instructions are shift right and shift left. They shift the value in A one bit to the right or one bit to the left. To implement these you need to make A a shift register with appropriate control signals.

Design a datapath that can Fetch, Decode, and Execute the given instruction set.

(a) **Design** a datapath that can Fetch, Decode, and Execute the given instruction set.

(b) Now **use** the datapath to calculate 1/5 the value in register R4 and store the result in R7. Use 13/64 as an approximation for 1/5 and ignore the issues of underflow and overflow (which are important, but this is not a computer arithmetic text per se, and we're just trying to get introduced to using these instructions right now so we can ignore for now some of the higher-level complexities.)

16.7 Design a datapath that can Fetch, Decode, and Execute the following instruction set. You may use registers, MUXes, comparators, register files, and ALU's in your design. (Remember that the ALU can perform all of the calculations; you do not need separate units for each arithmetic or logic operation.)

`MOV A, #N`	Move N into A
`MOV Rn, #N`	Move N into Rn
`MOV A, Rn`	Move Rn into A
`ADD A, Rn`	Add A + Rn and store the result in A
`SUB A, Rn`	Subtract A – Rn and store the result in A
`MUL A, Rn`	Multiply A x Rn and store the result I n A
`XOR A, Rn`	Calculate the bitwise XOR of A and Rn and store the result in A
`JZ N`	Jump to PC + N if A = 0

The last instruction is called a *branch*, or *jump*, instruction. It will change the order in which the instructions are executed, and is used for performing loops. Since the Program Counter (PC) holds the address of the next instruction to be fetched and executed, the branch instructions operate by changing the value in the PC. In this case, the `JZ N` instruction will change the value in the PC to PC + N (only when A = 0).

16.8 Design a datapath that can Fetch, Decode, and Execute the following instruction set. You may use registers, MUXes, comparators, register files, and ALU's in your design.

`MOV A, #N`	Move N into A
`MOV Rn, #N`	Move N into Rn
`MOV A, Rn`	Move Rn into A
`MOV Rn, A`	Move A into Rn
`ADD A, Rn`	Add A + Rn and store the result in A
`ADD A, #N`	Add A + N and store the result I n A
`ADD Rn, #N`	Add Rn + N and store the result in Rn
`XOR A, Rn`	Compute A XOR Rn and store the result in A
`XOR A, #N`	Compute A XOR N and store the result in A

16.9 Design a datapath that can Fetch, Decode, and Execute the following instructions.

MOV X, #N	Store the value N in register X
MOV Y, X	Copy the value from register X into register Y
MOV Rn, Y	Copy the value from register Y into Rn
ADD Rn, X, Y	Add X + Y and store the sum in Rn
SUB X, Rn, #N	Subtract Rn – N and store the result in X
XOR Y, X	Compute the bitwise XOR of Y and X and store the result in Y

Label all your control signals, use the values from the instruction encoding when possible, and do not include unnecessary connections. You may use all of our digital components, including a register file and a single ALU.

16.10 Consider the following instruction set:

`LD Rn`	`010bbnnn`	If `bb = 11`, load register nnn from a value in external memory. Otherwise, load register `nnn` from an address location equal to the sum of the PC and the least significant four bits of register `0bb`)
`LD A, #N`	`011xxxxx`	Load A with an 8-bit immediate value stored in code memory in the byte following this instruction
`MOV Rn, A`	`100xxnnn`	Load register `nnn` with the contents of A
`ALU Rn, op`	`101mmnnn`	A ← A op Rn, where op is given by `mm` and the register by `nnn`
`ALUi op`	`110mmxxx`	A ← A op X, where op is given by `mm` and X is an 8-bit immediate value stored in code memory in the byte following this instruction
`JMP addr, cond`	`111caaaa`	If A is zero and `c = 0`, or if `c = 1`, then jump to address `aaaa`

Design a datapath capable of fetching, decoding, and executing these instructions.

16.11 Consider the following instruction set:

`LD Rn`	`0100xnnn aaaaaaaa`	Rn ← M[aaaaaaaaa]
`ST Rn`	`0101xnnn aaaaaaaa`	M[aaaaaaaa] ← Rn
`ST (Rn)`	`0111xnnn`	M[(Rn)] ← A
`MOV A, #N`	`1000xxxx rrrrrrrr`	A ← rrrrrrr
`MOV Rn, #N`	`1001xnnn rrrrrrrr`	Rn ←rrrrrrr
`MOV Rn, A`	`1010xnnn`	Rn ← A
`ADD Rn`	`0000xnnn`	A ← A + Rn
`SUB Rn`	`0001xnnn`	A ← A - Rn
`NAND Rn`	`0010xnnn`	A ← A NAND Rn
`COMP Rn`	`0011xnnn`	A ← two's complement of Rn
`JMP`	`1100xxxx rrrrrrrr`	PC ← rrrrrrr
`JZ`	`1101aaaa`	If Z, PC ← PC + {sign extend}aaaa

16.12 Design a datapath which can implement the following instructions:

`LD A`	`110xxxxx`	Load A with the 8-bit value found in the byte following this instruction
`ST A`	`101xxxxx`	Store A in the 8-bit address location found in the byte following this instruction
`MOV Rn, A`	`101xxxnn`	Move the value in A to one of four general purpose registers
`MOV Rm, Rn`	`110xmmnn`	Move data from one register to another
`ADD A, Rn`	`001xxxnn`	Add the value in A to Rn and store result in A
`JMP Rn`	`010xxxnn`	If A = 0, then jump to the address found by adding the value in Rn to the PC

16.13 Design a datapath which can implement the following instruction set

`LD Rn`	`010xxnnn`	Load register `nnn` with an external input
`LD A, #N`	`011xxxxx`	Load A with an 8-bit immediate value stored in code memory in byte following this instruction
`MOV Rn, A`	`100xxnnn`	Load register `nnn` with the contents of A
`MATH Rn, op`	`101mmnnn`	A ← A op Rn, where op is given by `mm` and the register by `nnn`
`LOGIC Rn, op`	`110ggnnn`	A ← A op Rn, where op is given by `gg` and the register by `nnn`
`JMP addr, cond`	`111caaaa`	If A is zero and `c` = 0, or if `c` = 1, then jump to address `aaaa`

16.14 Consider the following branch instructions:

`bz rel`	branch to `rel` if the z bit in the status register is 1
`bgr ra, rb, rel`	branch to `rel` if the value in ra is greater than the value in rb
`bgm ra, N, rel`	branch to `rel` if the value in ra is greater than the value in memory location N
`bgi ra, #N, rel`	branch to `rel` if the value in ra is greater than the immediate value N

(a) Consider the `bgr` and `bgi` instructions. In what sense is `bgr` "better"? In what sense can `bgi` be considered "better?" Be specific.

(b) Rank order these four instructions based on how much hardware they require to calculate whether the branch is taken. Give explanations for your rankings, including a high-level overview of the hardware required.

Chapter 17
State Machines

The previous chapters showed how we can build digital circuits to implement any of the required computations for a particular algorithm. In order to handle the control structures, that is, the *ifs* and *fors* and *whiles*, in an algorithm, we need to consider state machines in addition to datapaths. This chapter introduces *stand-alone state machines* and the next chapter will talk about how we can link a state machine to a datapath to fully implement an algorithm in hardware.

Through the idea of **state** we seek to capture, understand, and ultimately control our digital systems. The **state of a system** is a collection of all the important details that determine exactly what is happening now that distinguishes the system *right here right now* from the system at other times. We've mentioned the idea of state already in the context of flip flops and counters. Now it's time to stretch the idea all sorts of directions and see how far it can take us.

A digital circuit that incorporates state at its heart is called a **state machine**. Our counters were simple examples of state machines with the current count encompassing the state. We had inputs such as increment and decrement that would transition the machine into a new state (count), and this new state was based not only on the increment/decrement input but also on the state the counter was currently in. To design a state machine is to articulate the response of the system to inputs where this response surely varies from one state to another state. The key is that a single given input may generate significantly different outputs depending on what state it happens upon at the exact moment of its encounter with the machine.

The general case involves two combinational logic systems we need to design to fully specify the state machine's behavior. The **next state logic** captures the state transition and the **output logic** specifies the signals the machine sends to any connected devices. Both the next state and output logic can depend on both or either of the input or current state. The **state memory** stores the label associated with the current state. The completed diagram of the generic state machine can be seen in Fig. 17.1.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_17

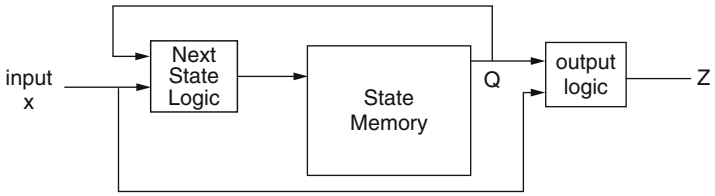


Fig. 17.1 General form of a state machine

Since we require one logic function per bit in the state and output, for simple machines with 5–8 states and a 2-bit output we're looking at 5 total equations to synthesize. Adding states and outputs increases this amount significantly. It's time to review the material on logic function synthesis before we continue, because each state machine design problem involves quite a few individual optimizations. We're going to be using the variable-entry K-map technique detailed in Chap. 5 on Advanced Logic Function Minimization at various points as we work with state machines for the rest of the text, so that may be a section worth reviewing as well.

Our task when constructing a state machine is to figure out what states are needed and to map out the transitions among them. That is, for each state we need to know what is the next state the machine will go to in response to each of the possible inputs to our system. It's common to represent this information visually in what is called a **state diagram.** We use circles to represent states and arrows the transitions. To each arrow we append an input. We can associate an output with either the transition or the state itself. In Fig. 17.2, we see examples.

In the case where we associate output with a state, we say we have a **Moore machine**. When we permit output to be a function of both state and input we say we have a **Mealy machine.** We'll often be asked to use Moore-type or Mealy-type outputs in our designs. Each approach has advantages and disadvantages in different contexts. Moore machines tend to require more states but can have simpler logic.

Let's build a state machine!

Sequence Detector

Perhaps the most basic non-trivial state machine we can put together is a **sequence detector.** This category of machine will receive an input stream of 1's and 0's and output a 1 when the past sequence of inputs matches the one (or one of many) the machine is designed to detect. We can tell this is a sequential logic device rather than a combinational logic device because sometimes an input of 1 will result in an output of 0 (the sequence has not yet been fully presented to the machine) and sometimes an input of 1 will result in an output of 1 (if the sequence to be detected ends in 1.) The internal **state**, then, is going to represent how much of the sequence

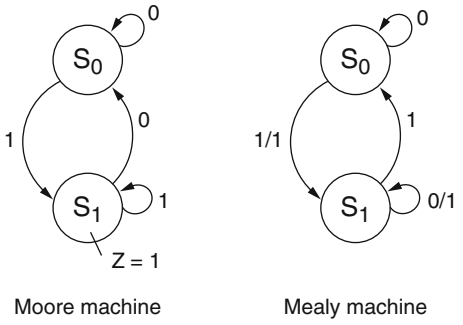


Fig. 17.2 Moore and Mealy machines

has already been detected and the output will depend on it. We'll use x for the input and z for the output.

Let's say we're going to recognize the sequence 110. What states will we require? Think about it for a second and then continue.

First, we'll need a default state. Often we start systems with such a state and almost always we name it the **idle state**. In this case the idle state indicates we have made no progress towards detecting the sequence. We'll start in this state and we'll actually return to it whenever we fail to progress in the sequence. Say we see a 1 followed by a 0 input. The 1 starts us on our way towards detecting our sequence but then the 0 ruins everything and we are now back where we started at the idle state.

Then we'll need states for the two 1's that lead off the sequence. Once we've detected that second 1 we're starting to get excited! The sequence is almost complete! We know we'll cry and transition back to idle if we see yet another 1 in the input. But what if we get 0? We'll have detected the sequence and are ready to emit a 1! But do we need a separate state for this?

It depends on the type of machine we are designing. If we use Moore-type outputs then we have to have a state set aside specifically for the $z = 1$ output. The state that detected the second 1 surely has $z = 0$ and with Moore-type outputs there can be only one output per state. But, with Mealy type outputs we can more finely tune our system. We can sometimes output $z = 1$ and sometimes output $z = 0$ from the state that detected the second 1. Because of this, we an associate the $z = 1$ output with the successful detection of the sequence when $x = 0$ and associate a $z = 0$ output with the failure to detect indicated when $x = 1$. Both inputs will lead to a transition back to the idle state, however, because whether or not we detect the sequence we'll be finished and ready to continue scanning input.

In diagram form, represented in Fig. 17.3, is the 110 sequence detector in both Moore and Mealy forms. We use the notation D1 to indicate the state wherein we've detected the first 1 in the sequence, D11 to indicate detection of the first two 1's, and D110 to indicate our eureka moment. We label the idle state simply, idle.

The first thing that jumps out at us is that the Mealy machine has only 3 states while the Moore machine requires 4. While that is true, it's not necessarily the case

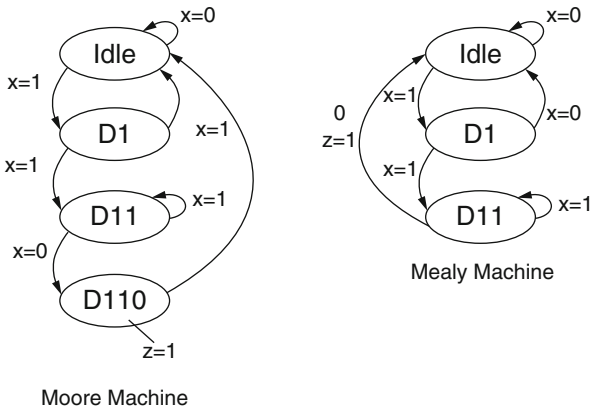


Fig. 17.3 State machines for the 110 sequence detector

that this means the Mealy machine is automatically a better choice for this application. The thing to keep in mind is that we require on state equation for each **bit** in the state representation, not for each state itself. So what matters is whether we're using the same number of bits in each case. Since both 3 and 4 states require 2 bits to represent them, it's not obviously clear that using 3 is more efficient than using 4 (though it will turn out to be that way in this case.) If we go to a 5th state, however, now we have to add an entire bit to the representation and that adds expense to the circuit. We want to be aware whenever we cross a power-of-two threshhold, because to go from 8 to 9 states or 16 to 17 states is a much bigger deal than going from 7 to 8 or from 17 to 18, say.

Another thing to note is the behavior of state D11 in both machines. When we see a 0 on the input we proceed with asserting $z = 1$. But what happens when $x = 1$? We end up remaining in the D11 state. This is because the state represents detecting the first two 1's in the sequence and if we detect a thousand 1's in a row the last two 1's still form the first two 1's in the 110 sequence. So once we get two 1's we can patiently (or impatiently) wait for the input to be $x = 0$ so we can complete this particular cycle.

We denote that the D110 state in our Moore machine outputs $z = 1$ by drawing a line from the state to the statement "$z = 1$." We could add similar notation for "$z = 0$" to the other states but we typically only concern ourselves with denoting **assertion** of signals. In this case, since our output is active-high, assertion of z means setting $z = 1$, so it's the only signal we show. As we proceed with state machine design we'll see that many state machines require multiple output signals. In these cases we'll be able to extend our notation by always clearly indicating what the variable name is for the particular output being asserted in a given state. That's why we go ahead and write out z here even though, because it's the only output we have, we could have just labeled the output as 1 without ambiguity. (Same story for why we label the inputs $x = 1$ and $x = 0$ throughout: some systems have multiple inputs and we want to get into the habit of reducing confusion by clearly labeling our signals.)

In the Mealy machine we see that the line to z = 1 comes from the arrow itself. This indicates that z = 1 is a function of both the fact that we're in state D11 and the fact that we're seeing input x = 1 while in that state. The other output from D11, corresponding to x = 0, has no such output designation.

Finally, note in the Moore machine the transition from D110 back to Idle has no label. This indicates that the transition always occurs regardless of input. We could list x = 0 and x = 1 there, but we don't need to. Many state machines will have these types of transitions and we may as well get used to them.

This is all really important! Make sure you can read through all this, grok the diagram, and tell the stories relayed herein. You simply must get the significance of the states, the transitions, the labeling of inputs and outputs, and the distinction between Moore and Mealy type outputs.

The state diagram, now that it took so long to generate and work through, is actually only the beginning of the design process. In fact, for sufficiently large state machines diagramic representations must either be severely truncated or abstracted or even altogether set aside for a better visual. For our simple machine, the diagram serves well enough the purpose of communicating the essence of the thing, but how do we use it to design? Where are the logic functions, the 1's and 0's and what about my beloved K-maps? How can I get K-maps from the diagram!

The answer is that we need to convert the diagram into a truth table. We can represent the same information found in the state diagram, how the system responds to inputs with state transitions and ouputs, in another form: the **state table.** We've already been introduced to a state table way back when we were building flip flops out of other flip flops. Now it's time to generalize to any state machine. In Table 17.1 you can see the state tables for our Moore and Mealy machines for the 110 sequence detector.

The first thing we need to do when building a state table is, like we do with everything else inside the computer, **encode the state**. So far we've used labels like Idle and D11 for the state, but ultimately the labels we associate with the states inside the computer must be logic-1's and logic-0's. So, like we've done with numbers and our instructions, we very much need to figure out a way to represent our states digitally. For this problem, and for much of the work we'll do actually, we will adopt the simple binary ordering for states. That is, we'll label the first state in our system, in this case Idle, with binary 0, the next state with binary 1, etc. It turns out that there are better ways to encode states but these are rather mathematically involved, pretty much ad hoc without any guarantee of success, and best handled by automated processes. They are discussed in the State Machine Optimization chapter, but for now we'll stick with a simple binary ordering.

Next, we put into the state table the same information we represent in the state diagram. We list the states in their encoded order and indicate what the next states and outputs are for each input. Notice in the Moore machine we have one input per state while in the Mealy machine we permit multiple inputs per state, depending on the input.

Take care to note the labels of the columns used here. The **present state** is always represented with Q's because that's the symbol we use to indicate the value

Table 17.1 State tables for the 110 sequence detector

Moore Machine State Table						
	Present state		**Input**	**Next state**		**Output**
	Q1	**Q0**	**x**	**D1**	**D0**	**z**
Idle	0	0	0	0	0	0
			1	0	1	
D1	0	1	0	0	0	0
			1	1	0	
D11	1	0	0	1	1	0
			1	1	0	
D110	1	1	0	0	0	1
			1	0	0	
Mealy Machine State Table						
	Present state		**Input**	**Next state**		**Output**
	Q1	**Q0**	**x**	**D1**	**D0**	**z**
Idle	0	0	0	0	0	0
			1	0	1	0
D1	0	1	0	0	0	0
			1	1	0	0
D11	1	0	0	0	0	1
			1	1	0	0
Unused	1	1	0	d	d	d
			1	d	d	d

stored in a flip flop. The **next state** is here represented by D's because we're going to use DFF's to implement the state memory in this design. Had we chosen to instead use an SR or JK or T flip flop, we would eschew the D labeling and append to the table the appropriate implementation columns as we saw in the Latches, Registers, and Timing chapter.

We have an **unused** state in the Mealy machine. This puts a lot of don't cares into the state table. We may care about making our machine **robust** (see the Counters chapter for a discussion of that) but often we'll just use the don't cares to best minimize our logic functions.

Other than that, you ought to be able to read the table and see how it tells the same story about the life of our 110 sequence detector as the state diagram. More than that, really: you should be able to look at a state table and figure out what the machine is doing!

So, our next state and output equations—the "design" of the state machine—are represented by the columns D_1, D_0, and z. To complete the design of our sequence detector we need to synthesize these functions.

Here are the relevant K-maps (Fig. 17.4):

Note that the output function for the Moore machine doesn't depend on the input x so we can use the two-value K-map with just Q_1 and Q_0 involved. (Actually, we'

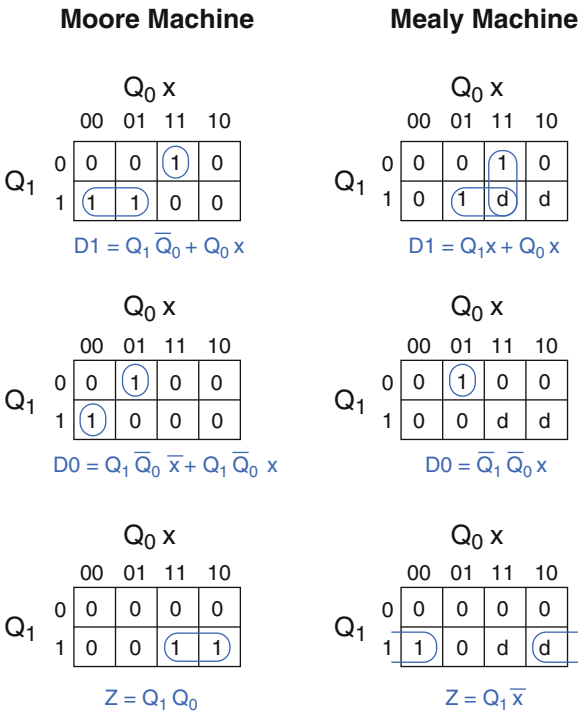


Fig. 17.4 K-maps for the next state and output equations specified in the state machines of Table17.1.

can just see its form from the table and don't need a K-map at all, but the principle holds: Moore machine output synthesis can ignore the input dimensions.)

We can then draw the circuit diagram for the system (see Fig. 17.5).

Sometimes we'll have a MUX and DEC to implement the next state and output equations, but here we've shown the gate-level implementation.

That's it! That's the design process for state machines. To summarize (Table 17.2):

We'll give a few more examples of state machine design here, then get into **datapath controllers** in succeeding chapters. These are specific kinds of state machines, so knowing how to build the basic designs introduced in this chapter will go a long way towards understanding them.

Detecting Two Sequences

Let's look at a more complicated sequence detector. What if we want to detect more than one sequence? Suppose we want to detect both the sequences 1001 and 1110? Furthermore, suppose we want our detector to be **nonrepeating**, that is, we want to be able to detect the sequences if they overlap.

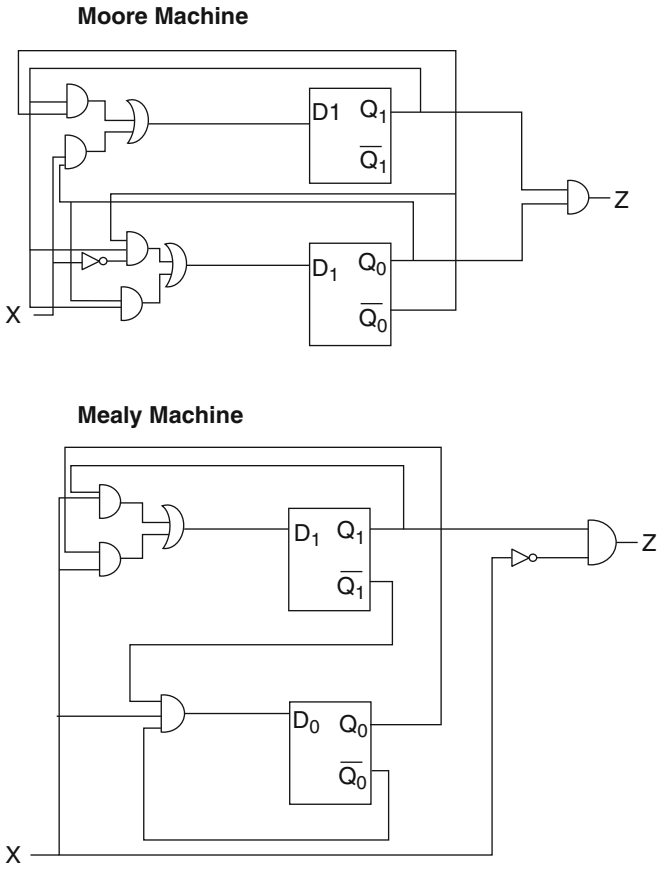


Fig. 17.5 Circuit diagrams for the state machines specified in Table 17.1.

Table 17.2 State machine design process

1. Decide what your states will represent
2. Work out the state transitions and outputs as responses to inputs
3. Draw the state diagram (if applicable)
4. Build the state table to specify the next state and output logic functions
5. Synthesize the next state and output functions
6. Implement

Here's an example input/output trace to illustrate the difference between repeating and nonrepeating detectors:

$$\begin{aligned} x &= 0\,1\,1\,0\,0\,1\,1\,1\,0\,1\,1\,0\,0\,1\,0\,1\,1\,1\,0 \\ z_{\text{repeating}} &= 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1 \\ z_{\text{nonrepeating}} &= 0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1 \end{aligned}$$

Using Moore-type outputs, what does the state diagram look like? Think about this and try to draw it out before reading on.

We start with our Idle state and then need all sorts of detect states: D1, D10, D100, and D1001 for the first sequence, and D1, D11, D110, and D1110 for the second sequence.

The first thing to notice is that the two sequences actually share the D1 state. There is no need to have separate D1 states for each sequence (nor could we really do that even if we tried.) This is an important revelation for state machine design—sometimes a single state can represent multiple ideas. The second thing to consider is that at some points during the progression through states we're going to go from one sequence to the other. If we're at D11 for the second sequence and we detect a 0, for example, we know the second sequence is busted. If we were just detecting 1110 then we'd return to Idle from D11 when the input is a 0. However, since we have a second sequence to keep track of we can see that although the full 110 doesn't fit anywhere, the last two bits we've seen are 10 and that actually is the beginning of our first sequence and so we'll transition from D11 to D10 upon seeing an input of $x = 0$. Wow. This is not all that easy to see the first time encountering this concept, even for those with solid logic and mathematics background. Like any skill, designing with state machines takes time spent on task to develop, so don't feel like it's too big a challenge if you struggle at first.

In Fig. 17.6, you can see the final state diagram for the 1001/1110 overlapping sequence detector with Moore-type outputs:

Go through this and make sure you follow, particularly the transitions between the states associated with each sequence. Let's look at the state table (Table 17.3).

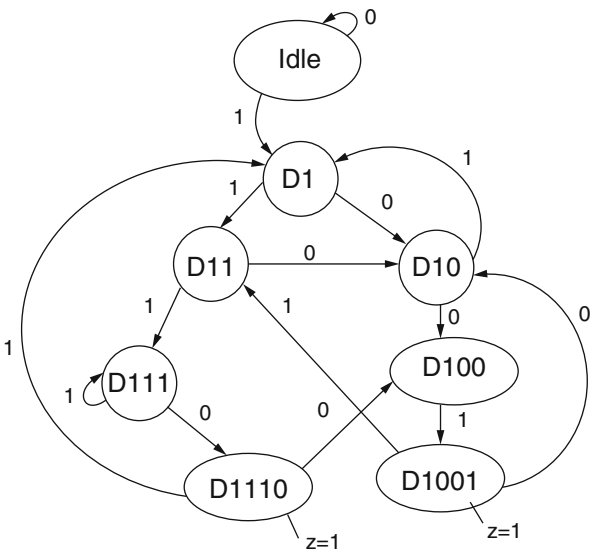


Fig. 17.6 State diagram for 1001/1110 sequence detector

Table 17.3 State table for the state diagram of Fig. 17.6

	Present state			Input	Next state			Output
	Q2	Q1	Q0	x	D2	D1	D0	z
Idle	0	0	0	0	0	0	0	0
				1	0	0	1	
D1	0	0	1	0	0	1	1	0
				1	0	1	0	
D11	0	1	0	0	0	1	1	0
				1	1	0	0	
D10	0	1	1	0	1	0	1	0
				1		0	1	
D111	1	0	0	0	1	1	0	
				1	1	0	0	0
D100	1	0	1	0	0	0	0	0
				1	1	1	1	
D1110	1	1	0	0	1	0	1	1
				1	0	0	0	
D1001	1	1	1	0	0	1	1	1
				1	0	1	0	

You can see this contains the same information as in the diagram, just encoded now in 1's and 0's. This lets us formally specify our three next state equations and our one output equation.

Synthesizing the functions now require four-variable K-maps for the minimization. Try your hand at them.

It's clear that state machine logic synthesis can quickly get unwieldy because it's quite easy for the number of input variables to grow rapidly. All it takes is a few more states and inputs and suddenly you're staring down a 9 or 10 variable map. Moving forward in this text we'll be using the variable-entry K-map option presented in the Advanced Logic Function Minimzation chapter for these state machines. This will allow us to use 3- and 4- variable maps to more readily find the minimal forms of our next state and output equations. It's going to be particularly useful because in many of our designs we'll have what is called a **sparse** variable: one that is is 0 for the vast majority of states and only comes into play in a few places in the diagram. It's much easier to isolate the effects of such inputs on the overall state machine design using a variable-entry K-map than it would be to add more dimensions only to populate them with mostly with 0's.

For the variable-entry map version of this sequence detector, we will reduce the input x from the state table and incorporate it into the maps themselves.

We get the same results with smaller maps. Make sure you can follow how the variable-entry works and, in addition, it's worth reworking this problem as a Mealy machine. If you can do that, then you're ready to move on.

Table 17.4 Implementation table for D, SR, JK, and T flip flops

Q_t	Q_{t+1}	D	S	R	J	K	T
0	0	0	0	d	0	d	0
0	1	1	1	0	1	d	1
1	0	0	0	1	d	1	1
1	1	1	d	0	d	0	0

Other Flip Flops for State Memory

While we usually use DFF's for the state memory, it's worth taking a little bit of time to work an example of using SR, JK, and T flip flops.

Recall in Table 17.4 the implementation table for the various flip flops from Chap. 11:

When working on a state machine design using flip flops other than DFF's we have to remember that the **next state** logic functions specify **those signals required to affect the desired transition**. For DFF's, those signals and the next states themselves are one and the same. But, for the other flip flops they are not. Instead, the next state logic functions we need to specify reflect the specific nature of the flip flop under consideration as summarized in our implementation table. It's a good time to review the Flip Flops chapter if you have forgotten how we derived this table.

Let's look at our 110 sequence detector from earlier and show, in Table 17.5, the full Moore machine state table for all four flip flops:

With all the don't cares running about in the SR and JK implementation specifications, it usually turns out that you get simpler next state logic with these flip flops. For special types of problems, such as binary counters, the T devices can also realize better logic than the DFF's. Make sure you can build the implementation table for state machines as shown above, as it may come in handy at some point. It also reinforces the operational principles behind SR, JK, and T flip flop.

Exercises

17.1 Design a 1011 sequence detector. Use (a) Mealy-type outputs and (b) Moore-type outputs.

17.2 Design a state machine that can detect either 1101 or 1010 sequences. Use (a) Mealy-type outputs and (b) Moore-type outputs.

17.3 Design a state machine that will analyze an input sequence and output 1 every third 1 that is input.

X = 01100101011100010101
Z = 00000100001000000100

Table 17.5 Full implementation table for the 110 sequence detector

	Present state		Input	Next state		Output	Implementations									
	Q1	Q0	x	D1	D0	z	S1	R1	S0	R0	J1	K1	J0	K0	T1	T0
Idle	0	0	0	0	0	0	0	d	0	d	0	d	0	d	0	0
			1	0	1		0	d	1	0	0	d	1	d	0	1
D1	0	1	0	0	0	0	0	d	0	1	0	d	d	1	0	1
			1	1	0		1	0	0	1	1	d	d	1	1	1
D11	1	0	0	1	1	0	d	0	1	0	d	0	1	d	0	1
			1	1	0		d	0	0	d	d	0	d	0	0	0
D110	1	1	0	0	0	1	0	1	0	1	d	1	d	1	1	1
			1	0	0		0	1	0	1	d	1	d	1	1	1

Table 17.6 Truth table for exercise 17.6

Present State		Input	Next State		Output	Activation			
		c			z	S1	R1	S0	R0
0	0	0	0	0	0				
		1	0	1	0				
0	1	0	1	0	1				
		1	1	1	0				
1	0	0	0	1	0				
		1	1	1	1				
1	1	0	0	0	1				
		1	0	1	1				

Table 17.7 Truth table for exercise 17.7

Present State		Input	Next State		Output	Activation			
		s			z	J1	K1	J0	K0
0	0	0	1	0	1				
		1	0	1	0				
0	1	0	0	0	1				
		1	1	1	0				
1	0	0	0	0	1				
		1	1	0	1				
1	1	0	0	1	1				
		1	1	0	0				

17.4 Design a state machine that will output a 1 whenever it detects at least two 0's in the three bits following two successive 1's. When it detects the two 1's it should scan the next three bits before resetting to look for the 1's again. Also, the output 1 should coincide with the third bit after the two 0's.

X = 010110011001011011011110
Z = 000000010000000000100000

17.5 Design a state machine to control a robot. It has states Idle, Search, Return, and Zap. It has inputs Sensor 1, Sensor2, LowBat, and Go. It will begin in Idle until Go is received, in which case it will go to Search. When LowBat is asserted it will Return. When Sensor 1 and Sensor 2 are asserted at the same time, it will Zap and then go back to Searching.

17.6 For the state machine given in Table 17.6, (a) draw the state diagram, (b) fill in the activation columns, (c) calculate the minimal next state logic, and (d) calculate the minimal output logic. Use SR Flip-Flops for the state memory.

17.7 For the state machine given in Table 17.7, (a) draw the state diagram, (b) fill in the activation columns, (c) calculate the minimal next state logic, and (d) calculate the minimal output logic. Use JK Flip-Flops for the state memory.

17.8 For the state machine given in Table 17.8, (a) draw the state diagram, (b) fill in the activation columns, (c) calculate the minimal next state logic, and (d) calculate the minimal output logic. Use T Flip-Flops for the state memory.

Table 17.8 Truth table for exercise 17.8

Present State		Input s	Next State		Output z	Activation T1	Activation T0
0	0	0	0	1	0		
		1	0	0	0		
0	1	0	0	1	0		
		1	1	0	0		
1	0	0	0	1	0		
		1	1	1	0		
1	1	0	0	1	1		
		1	0	0	0		

Table 17.9 Truth table for exercise 17.13

	Present state			Inputs		Next state			Outputs				
	Q2	Q1	Q0	G	S	D2	D1	D0	LD	dec	LDF	set	valid
Idle	0	0	0	x	0	0	0	0	0	0	0	0	0
				x	1	0	0	1					
Init	0	0	1	x	x	0	1	0	1	0	0	1	0
Comp	0	1	0	0	x	1	0	0	0	0	0	0	0
				1	x	0	1	1					
Update	0	1	1	x	x	0	1	0	0	1	1	0	0
Valid	1	0	0	x	x	0	0	0	0	0	0	0	1
	1	0	1	**x**	**x**	x	x	x	x	x	x	x	x
	1	1	0	**x**	**x**	x	x	x	x	x	x	x	x
	1	1	1	**x**	**x**	x	x	x	x	x	x	x	x

17.9 Design an SR Flip-Flop out of a D Flip-Flop.
17.10 Design a T Flip-Flop out of an SR Flip-Flop.
17.11 Consider an AB flip flop which functions as follows:

A	**B**	**Function**
0	0	set
0	1	hold
1	0	toggle
1	1	reset

Build an AB device out of a T flip flop.

17.12 Design a binary up/down mod 6 counter. If the input $c = 0$ then count up, and if $c = 1$ then count down. Use don't cares for unused states. Fill in the truth table and calculate the next state logic. Use SR flip flops for the state memory.
17.13 Calculate the minimal logic for the next state and output equations for the state machine specified in Table 17.9.
17.14 Draw the Moore state diagram and state table for a 10,010 non-resetting sequence detector.

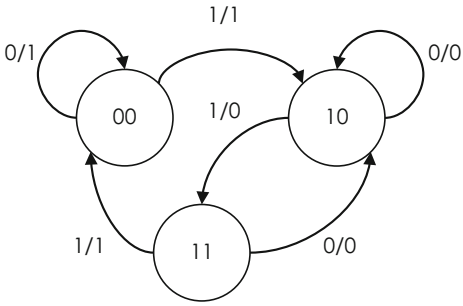


Fig. 17.7 State diagram for exercise 17.18

17.15 Build an S-R flip flop using only a toggle flip flop and combinational logic.

17.16 Suppose you are given specifications for an odd sort of memory element, the HS flip flop. Design this flip flop using only a DFF and combinational logic.

HS	**Action**
00	reset
01	set
10	reset
11	toggle

17.17 Draw a Moore machine state diagram for a non-resetting sequence detector that outputs a 1 if it detects either 1010 or 0110.

17.18 (a) Fill in the state table for the Mealy diagram in Fig. 17.7.
(b) Calculate two sets of next state equations: one using DFF's and one using S-R flip flops.
Compare the two sets of next-state equations you obtained.
(c) In what sense may it be better to implement this design using DFF's?
(d) In what sense may it be better to use SR-flip flops?

17.19 You've been handed specifications for two designs, the CW-flip flop and the HS-flip flop.

CW	**Action**	**HS**	**Action**
00	toggle	00	unused state
01	reset	01	toggle
10	set	10	hold
11	hold	11	reset

(a) Write the excitation tables for each of these flip flops:
(b) Draw the logic diagrams for each of the following four implementations:

1. Implement the CW flip flop using the HS flip flop.
2. Implement the HS flip flop using the CW flip flop.
3. Implement the CW flip flop using a JK flip flop.
4. Implement the HS flip flop using a toggle flip flop.

17.20 Consider a 1010 sequence detector. Draw the state diagram and state table for this machine using four different formats: resetting Moore, nonresetting Moore, resetting Mealy, nonresetting Mealy.

17.21 Consider a 110 sequence detector. Go through all the steps of state machine design (state diagram, state table, state assignment, encoded state table, next state equations, output equations, design implementation) for this device twice: use D flip flops the first time and use SR flip flops the second time. It is easiest to do this using a Moore machine, although you may use a Mealy machine if you want.

17.22 Draw the Mealy diagram for a non-resetting detector that outputs a 1 for either a 110 sequence or for a 101 sequence.

17.23 Consider a 010 sequence detector. Go through all the steps of state machine design (state diagram, state table, state assignment, encoded state table, next state equations, output equations, design implementation) for this device twice: use D flip flops the first time and use JK flip flops the second time. Work this using a Mealy machine.

17.24 Design an S-R flip flop using a D flip flop. Draw the resulting logic.

17.25 Design an S-R flip flop using a toggle flip flop. Draw the resulting logic.

17.26 Make a JK flip flop out of the T flip flop below. Show all your work and draw the resulting circuit.

17.27 Fill in the appropriate values for the two SR-latches in the state table shown in Table 17.10.

17.28 For the state machine described by Table 17.11, draw the state diagram, fill in the implementation columns in the state table, calculate the next state logic, and calculate the output logic. Use SR flip flops for the state memory.

Table 17.10 State table for exercise 17.27

Q_1	Q_2	NS		S_1	R_1	S_0	R_0
0	0	1	0				
0	1	1	0				
1	0	0	0				
1	1	0	1				

Table 17.11 State table for exercise 17.28

Present		Input	Next		Output	Implementation			
State		c	State		z	S1	R1	S0	R0
0	0	0	0	0	0				
		1	0	1	0				
0	1	0	1	0	1				
		1	1	1	0				
1	0	0	0	1	0				
		1	1	1	1				
1	1	0	0	0	1				
		1	0	1	1				

17.29 Design a state machine which will detect a 1011 sequence. It should output a 1 only when the sequence has been detected and then reset itself to scan for a new 4-bit sequence. Use Mealy-type outputs (make the outputs functions of the current state and input.) Draw the state diagram, fill in the state table, and calculate the next state and output logic. Use T flip flops for the state memory.

17.30 Design a state machine will will detect a 101 sequence. Make the output active-low: output a 0 if and only if the sequence has been detected. Use Moore type outputs (that is, the output is a function of the current state only.) Draw the state diagram, fill in the state table, and calculate the next state and output equations. Use JK flip flops for the state memory.

17.31 For the state machine described by Table 17.12, draw the state diagram, fill in the implementation columns in the state table, calculate the next state logic, and calculate the output logic. Use JK flip flops for the state memory.

17.32 For the state machine described by Table 17.13, draw the state diagram, fill in the implementation columns in the state table, calculate the next state logic, and calculate the output logic. Use T flip flops for the state memory. Also explain what this device does.

17.33 Draw the state diagram for a **Mealy** machine which can detect **both** of the sequences 1010 and 1100. That is, whenever the last four inputs are either 1010 **or** 1100 the output will be a 1. The output will be 0 otherwise. It may be helpful to label your states and list what they represent.

Table 17.12 State table for exercise 17.31

Present	Input	Next	Output	Implementation			
State	*s*	State	*z*	J1	K1	J0	K0
0 0	0	1 0	1				
	1	0 1	0				
0 1	0	0 0	1				
	1	1 1	0				
1 0	0	0 0	1				
	1	1 0	1				
1 1	0	0 1	1				
	1	1 0	0				

Table 17.13 State table for exercise 17.32

Present	Input	Next	Output	Implementation		
State	*s*	State	*z*	T1	T0	
0 0	0	0 1	0			
	1	0 0	0			
0 1	0	0 1	0			
	1	1 0	0			
1 0	0	0 1	0			
	1	1 1	0			
1 1	0	0 1	1			
	1	0 0	0			

Table 17.14 State table for exercise 17.37

Present State	Input c	Next State	Output z	Implementation S1	R1	S0	R0
0 0	0	0 0	1				
	1	0 1	1				
0 1	0	0 0	1				
	1	1 0	0				
1 0	0	0 1	0				
	1	1 1	1				
1 1	0	1 0	0				
	1	0 1	1				

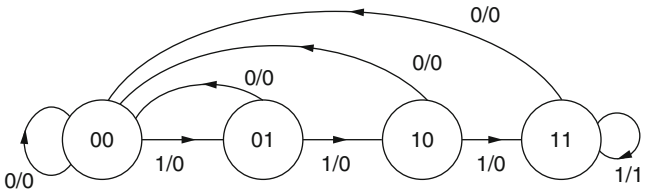


Fig. 17.8 State diagram for exercise 17.38

17.34 Draw the state diagram for a **Moore** machine which can detect **both** of the sequences 110 and 010. That is, whenever the last three inputs are either 110 **or** 010 the output will be a 1. The output will be 0 otherwise. It may be helpful to label your states and list what they represent.

17.35 A soda machine dispenses sodas for $1.25. The machine accepts nickles, dimes, and quarters. When enough coins have been inserted it dispenses the soda and returns any necessary change. Design the state machine controller for this soda machine. Its inputs should be *N, D,* and *Q* depending on the coin type, and its outputs should be *Dispense* which dispenses the soda and *outN, outDime,* and *outTwoDimes* depending on how much change is required. (Assume only a single coin may be input per state/cycle.)

17.36 For the state machine described by Table 17.14, draw the state diagram, fill in the implementation columns in the state table, calculate the next state logic, and calculate the output logic. Use SR flip flops for the state memory.

17.37 Given the state transition diagram in Fig. 17.8, answer the following questions.

(a) Construct the state table from the state transition diagram. Use D flip-flops for the state variables. Label the input and output variables and the state variables on the state transition diagram.

(b) Draw the sequential circuit (state machine) based on the state table that you derived from **part a**.

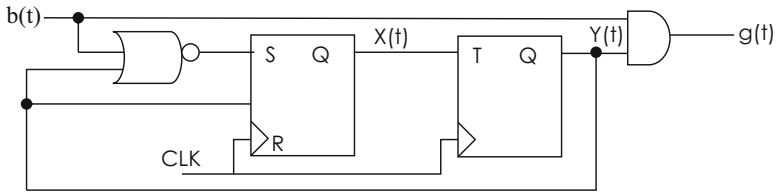
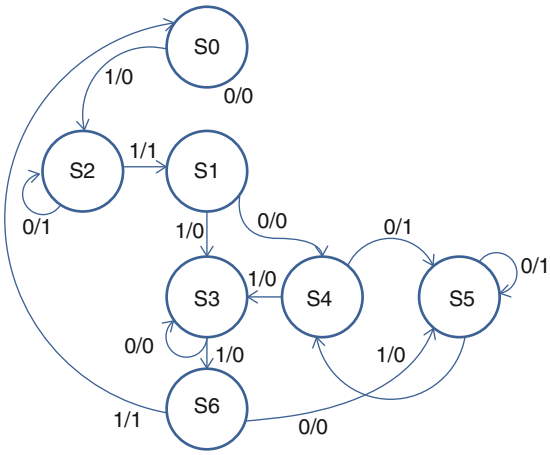


Fig. 17.9 State machine circuit diagram for exercise 17.39

Fig. 17.10 State diagram for exercise 17.40



17.38 Given the sequential circuit in Fig. 17.9 with the state variables X(t) and Y (t), externally applied input b(t) and output g(t). Answer the questions for **parts a and b** below.
(a) Fill in the missing values for the state table. **Show all relationships** used to fill in the state table values for *full credit*.
(b) Draw the state transition diagram based on the state table above.

17.39 Draw the state table for the state diagram in Fig. 17.10. Use a binary encoding (S0 = 000, S1 = 001, etc.) Be sure to include Present State, Next State, and Output columns. You may organize your table any way that makes sense to you.

17.40 Fill in the J1 and K1 columns in the state table given in Table 17.15:

17.41 Design a state machine which will detect a 0111 sequence. It should output a 1 only when the sequence has been detected and then reset itself to scan for a new 4-bit sequence. Use Mealy-type outputs (make the outputs functions of the current state and input.) Fill in the state table, calculate the next state logic for T1, and calculate output logic. Use T flip flops for the state memory.

17.42 For the state machine described by Table 17.16, draw the state diagram and fill in the implementation columns in the state table for both T and JK flip flops.

Table 17.15 State table for exercise 17.41

Present		Input	Next		Output		
State		c	State		z	J1	K1
0	0	0	0	1	0		
		1	1	1	0		
0	1	0	1	0	1		
		1	1	1	1		
1	0	0	1	0	1		
		1	1	1	0		
1	1	0	0	0	0		
		1	1	1	1		

Table 17.16 State table for exercise 17.43

Present		Input	Next		Output	JK Implementation			T Implementation		
State		c	State		z	J1	K1	J0	K0	T1	T0
0	0	0	0	0	1						
		1	1	0	1						
0	1	0	1	0	0						
		1	0	0	0						
1	0	0	0	1	1						
		1	1	1	1						
1	1	0	0	1	0						
		1	1	0	0						

17.43 Draw the state diagram and state table for a non-resetting 1011 sequence detector.

17.44 Construct state machine for a sequence recognizer having a single input variable x and two output variables z1 and z2. The output z1 is asserted if and only if exactly two of the three bits following a “010” sequence are 1’s and the output z2 is asserted if and only if all the three bits following a “010” sequence are 1’s. Upon completing the analysis of the three bits following the “010” sequence the network is to reset itself and start over.

Chapter 18
Datapath Controllers

This is it! This is the chapter that really conveys how algorithms can be implemented fully in digital electronic circuitry. It's in a real way the climax of the entire text, whose name Digital Logic for Computing really communicates the idea that we're focusing herein on the implementation of computing, that is, algorithms, using the techniques of digital design.

This chapter combines the previous two on datapaths and state machines in that we're now going to go beyond the stand alone state machine and build some that connect to datapaths. We'll complete some examples that were started in the Datapaths chapter and go through new ones from scratch as well. This chapter really gets into the heart of how to interpret the notion of *state* in the context of *algorithm*.

It also touches on a very prevalent and successful digital design philosophy—the partitioning of systems into datapaths and controllers is widely used and quite effective. It helps bring together the notions of generic algorithms and general purpose processors in that we can see the algorithms at their most basic computations and then see the individual instructions in the processors as the discrete algorithmic elements that they really are. So, sit back, read well, and above all, *enjoy* this chapter. For the student or professional interested in computing, this chapter has so much to offer.

States and Algorithms

The first thing to understand is the way we use states in these hardware systems. It's not a flowchart! Sometimes programmers are used to thinking in terms of flowcharts and, especially because state diagrams can sometimes resemble these charts, we need to remember that the technology of states requires us to relate them to variables in a particular way. This is of utmost importance in setting up the state machines that implement our algorithms.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_18

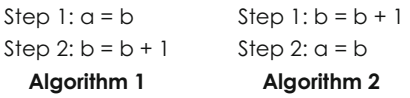


Fig. 18.1 General computations

It's critical in this business to always remember that the registers in the datapath and the register storing the state are attached of the **same clock signal** and therefore update all at the same time. Therefore, when determining what steps of the algorithm can be in the same state we have to remember this rule of registers. We can use a register value and update it in the same cycle, but we cannot update a value and then use the new value in the same state. For example, consider the following two general computations shown in Fig. 18.1.

In the first algorithm we are assigning the current value of b to a and then incrementing b. At the end of these two steps a and b should hold different values. We can combine both of these assignment statements into a single state, because the three register updates required, (1) a receiving b, (2) b receiving b + 1, and (3) the state register updating to the next state, all occur simultaneously and correspond with the logic of the algorithm. So, even though this algorithm seems to have two steps we can see that it only requires a single state to implement these steps. As we connect our algorithms to state machines, we'll be making these determinations.

In contrast, the second algorithm is necessarily sequential. The second step is dependent on the fulfillment of the first. We can't update a until b is incremented because the logic of this algorithm demands that a and b have the same value, that of the updated b, at the conclusion of the two steps. In this case, we must implement each step in a separate state.

So, we have the situation where very similar two-step algorithms are implemented in differing numbers of states. We have to keep this thought process in mind constantly as we seek to structure the state machines. In fact, it is this part—the setting up of what each state is meant to accomplish—that is pretty much the totality of the human designer's impact on the entire design! We have computation tools to do all further synthesis and simplification! It is the specification of the state machine implementation of the algorithm itself (and, of course the creation of the algorithm in the first place) that represents the craft and maybe the art of the human designer.

This concept in mind, we'll now continue with several case studies of how to build controller-datapath systems.

Example: Greatest Common Divisor

Recall the GCD algorithm and datapath from the Chap. 15 (Fig. 18.2).

```
input unsigned integers x and y
while ( x != y )
   if ( x > y )
       x = x - y;
```

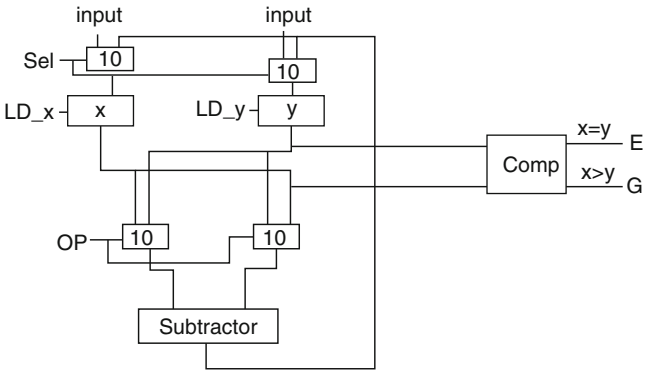


Fig. 18.2 GCD calculator datapath

```
    else
        y = y - x;
assert valid
```

Here we see the electronic circuit that is capable of performing the four necessary computations for this algorithm: x – y, y – x, x! = y, and x > y (the latter two computations being boolean expressions.) We've put off until now the question of how to sequence these computations. Armed now with the idea of *state* from the previous chapter, we can associate each step of the algorithm with a state.

In the first state we'll input the values x and y. It's interesting here to compare to the sequential programmer's mindset where we would never put a `get(x)` and a `get(y)` statement on the same line. In hardware we get a fair degree of parallelization for free and since the registers x and y are independent, we can certainly update both of them in the same state.

The next state requires some consideration. We are entering the while loop and we need to evaluate the boolean expression x! = y. If this were a flowchart we'd just draw arrows from the input block to one of two next blocks to indicate whether we enter the loop or proceed to the last statement in the algorithm. But this is not a flowchart! We in fact need to wait a state before we have access to the x! = y signal! Remember the rules regarding registers: we cannot update a register and use the new value in the same state. So we're looking at the following two steps here:

Step 1: x = new value

Step 2: x != y

This mirrors Algorithm 2 from the previous section in that step 2 depends on the value in step 1 being updated. This is where looking at the boolean expressions in the while loop as actual expressions to be evaluated really comes in handy. All your programming experience can be used here in hardware design.

Putting this all together means we must wait until the state after the input state to see the evaluation of x! = y. Now, what about the next computation? The next thing

to evaluate is actually the x > y in the if statement. This, now, is an independent computation! Therefore, we can actually access the values of x! = y and x > y in the same state and make our decisions on both the while loop and the if statement at the same time in the same state! Already, then, within the first two states of this algorithm we're seeing a powerful type of concurrency, virtually for free, of a sort that software engineers pay dearly for.

In Fig. 18.3 we see the beginning of the state machine diagram for our controller beginning to emerge.

The comp state looks at both E and G status signals from the datapath representing the evaluation of the loop and selection statement's conditional expressions. We can see that there are three transitions coming from the comp state. These transitions correspond to the remaining three steps in the algorithm. Since the while loop is bypassed completely in the event x = y, the E = 1 input doesn't also need a G component to it in the diagram. So, while it may seem awkward to see three of something in a digital design, these transitions here do make sense and fit in with what the algorithm is doing.

All that remains is to finish up the diagram with the remaining states. We do this in Fig. 18.4. These are straightforward and simply correspond to the required computations as detailed in the algorithm.

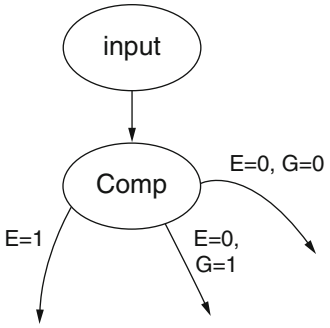


Fig. 18.3 Beginning of state machine controller for the GCD datapath

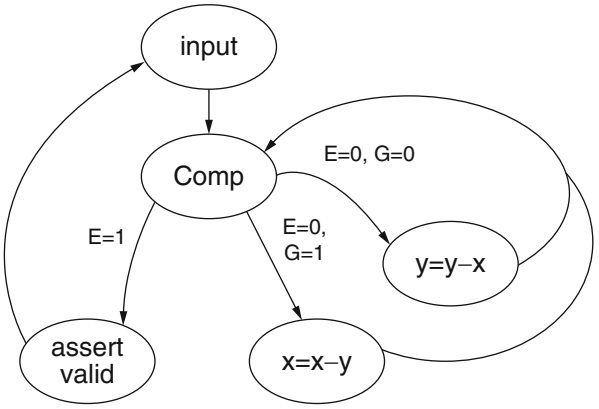


Fig. 18.4 Finalized GCD datapath controller state diagram

Table 18.1 Control word table for the GCD datapath

State	LD_x	LD_y	sel	op	valid
input x and y	1	1	1	d	0
x = x – y	1	0	0	1	0
y = y – x	0	1	0	0	0
assert valid	0	0	d	d	1

Table 18.2 State table for the GCD datapath controller

	Present state			Input		Next state			Outputs				
	Q2	Q1	Q0	E	G	D2	D1	D0	LD_X	LD_Y	Sel	op	valid
input	0	0	0	d	d	0	0	1	1	1	1	d	0
comp	0	0	1	0	0	0	1	1					
				0	1	0	1	0	0	0	d	d	0
				1	d	1	0	0					
X = X–Y	0	1	0	d	d	0	0	1	1	0	0	1	0
= y–1	0	1	1	d	d	0	0	1	0	1	0	0	0
asse valid	1	0	0	d	d	0	0	0	0	0	d	d	1

The x = x –y and y = y – x states both transition back to the comp state without dependence on any inputs. The assert valid state is shown here returning to the input state. We could potentially add in an idle start state like we used in the sequence detectors or some other algorithms where we wait on a start signal S to proceed, but this is a fine rendition of the algorithm for our current purposes.

While in the sequence detector state machines we'd draw the Z = 1 output in the appropriate state, here we suppress output notation from the diagram completely. We could add the asserted outputs to each state, but it can quickly overtake the diagram and render it unreadable. To detail the outputs, we'll rely on the control word table we constructed previously (see Table 18.1).

Now that we have the actual state diagram drawn, we can go ahead and follow our state machine design process and complete this controller. We'll encode the states in normal binary order for simplicity's sake; a later chapter will investigate optimizations for state assignment.

In Table 18.2 we see the full state table for the design. The outputs follow the control word table with the addition of the comp state which wasn't obviously needed when we first visited this algorithm in Chap. 15. We can see the inputs are sparse in that E and G are only relevant for transitions in the comp state and we liberally sprinkle the rest of the table with don't cares. We're using DFF's for the state memory.

For completeness, Fig. 18.5 presents, in impressive fashion, the K-maps to compute the final next-state and output equations for this controller.

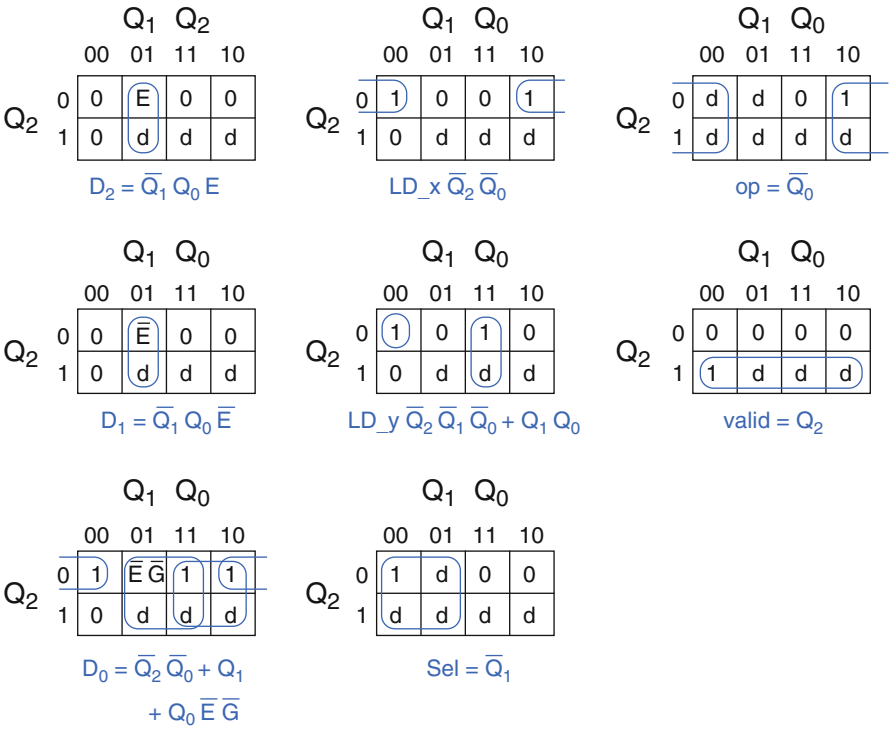


Fig. 18.5 K-maps for the next state and output equations for the GCD datapath controller specified in Table 18.2

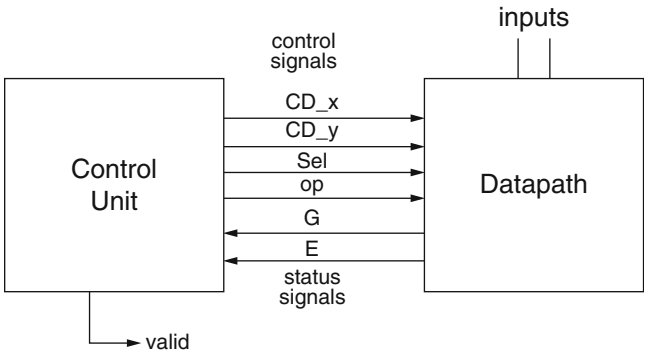


Fig. 18.6 Block diagram for the GCD datapath and controller

And the final block diagram of the entire controller-datapath system for the GCD calculator is seen in Fig. 18.6.

And there we go! A full algorithm implemented using datapath components and a state machine. You can keep going using the logic equations for the next state and output logic for the state machine and all the interior gates involved.

Example: Compute Factorial

For our next example, let's implement an algorithm that inputs a number n and outputs its factorial n!. The algorithm is as follows:

1. wait for start bit S
2. input n
3. product =1
4. product = product * n
5. n = n – 1
6. if n = 1, assert valid and return to 1
   else, return to 4

Notice we write this algorithm in a sequence of steps instead of pseudocode. The point is that it doesn't matter what form our algorithms take—as long as the steps are made clear we can envision them either way. For some designers with a strong programming background the pseudocode may be easier, but for others knowing these can be broken down into discrete steps in this manner is helpful. Either way, we proceed with the design in the same manner.

First, let's build the datapath. We need to go step-by-step and identify the components needed as well as the computations to be performed. Overviewing the algorithm we see we need registers to store n and product, a comparator to comput n = 1, and a multiply unit. We can store n in a counter with a decrement input and either equip the product register with a set control or multiplex its input. The result can be seen in Fig. 18.7.

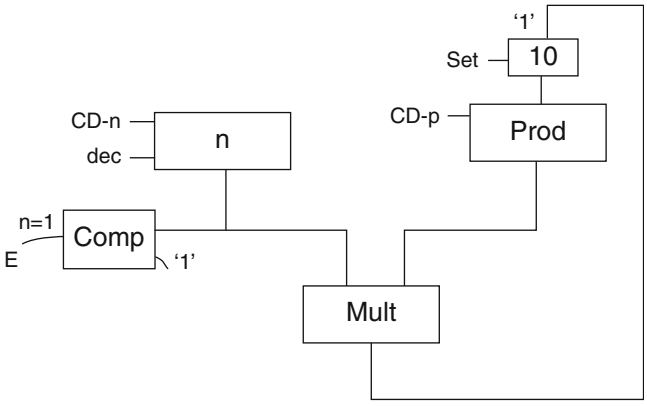


Fig. 18.7 Datapath for factorial calculator

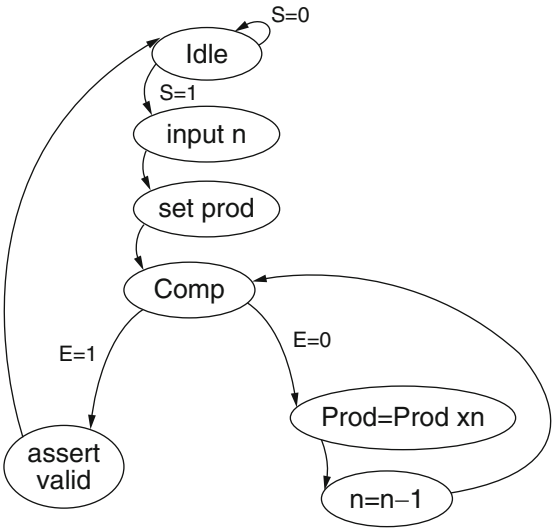


Fig. 18.8 First attempt at a state diagram for the datapath controller for the factorial calculator datapath of Fig. 18.7

Stop reading here and see if you can come up with the state machine controller for this datapath. We're going to go through the controller in multiple steps until we get to the final version. In Fig. 18.8, let's look at a first pass state diagram that assign each step of the algorithm to a single state in the controller.

The seven states here correspond to the six steps in the algorithm plus an additional comp state for the evaluation of the boolean expression $n = 1$. It's placement may seem curious as it doesn't appear in the algorithm until the final step, but if you look at it from the pseudocode perspective the body of the algorithm becomes

```
while ( n > 1 )
{
    prod = prod * n;
     n = n - 1;
}
```

From this it's clear we're checking the $n = 1$ (or $n > 1$ depending on how you look at it) condition early on in the algorithm. We could consider the `do. while` programming idiom and place the comp state after the $n = n - 1$ state, but the overall execution of the algorithm remains the same either way. In fact, the diagram given successfully handles the case where the input is $n = 1$ to begin with. If we first decrement n and then check whether it's equal to 1 we could get erratic behavior. The point here being that there are multiple ways to approach the algorithmic description and implementation of a given system and all one's expertise from either an engineering or a programming background can be brought to bear on these precise points.

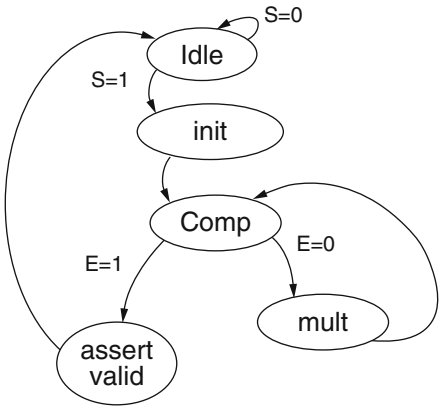


Fig. 18.9 Fixed comp calculation

Table 18.3 Control word table for the factorial state machine of Fig. 18.9

State	LD_n	LD_p	set	dec	valid
idle	0	0	d	0	0
init	1	1	1	0	0
comp	0	0	d	0	0
mult	0	1	0	1	0
assert	0	0	d	0	1

Let's take a careful look at these states, however. While we're not really focusing on state machine optimization in this chapter, it's worth tackling low-hanging fruit, and there are some easy to see efficiencies in this design.

Remember our rules for what things can be performed in a given state: it's all about the independence or dependence of register updates. We can see in the input n and set prod states that there is no conflict here. Both registers can be updated at the same time without consequence. We can combine these into a single initialization state. Similarly, the prod = prod * n and the n = n – 1 states can be made into one. This is a little less obvious because it relies on understanding the change to n won't take effect until the state transition so that the value being used in the prod update is indeed the old value of n, as intended. This is similar to the Algorithm 1 example from the beginning of the chapter. Again, make sure you understand the underlying logic behind these efficiencies. The new state diagram becomes that of Fig. 18.9.

The next step is to construct the control word table (Table 18.3) defining what control signals are to be generated in each state.

Not much new here. We can see don't cares in the MUX select line as we'd expect when the register it's feeding is not in use. The LD signals, as is customary, are turned off when we're not updating the related variables.

The completed state table for this state machine is given in Table 18.4.

The K-maps for the next state and output logic are given in Fig. 18.10.

And the final controller-datapath block diagram is in Fig. 18.11.

Table 18.4 State table for the datapath controller specified in Fig. 18.9 and Table 18.3

	Present state			Inputs		Next state			Outputs				
	Q2	Q1	Q0	S	E	D2	D1	D0	LD_n	LD_p	set	dec	valid
idle	0	0	0	0	d	0	0	0	0	0	d	0	0
				1	d	0	0	1					
init	0	0	1	d	d	0	1	0	1	1	1	0	0
comp	0	1	0	d	0	0	1	1	0	0	d	0	0
				d	1	1	0	0					
mult	0	1	1	d	d	0	1	0	0	1	0	1	0
assert	1	0	0	d	d	0	0	0	0	0	d	0	1

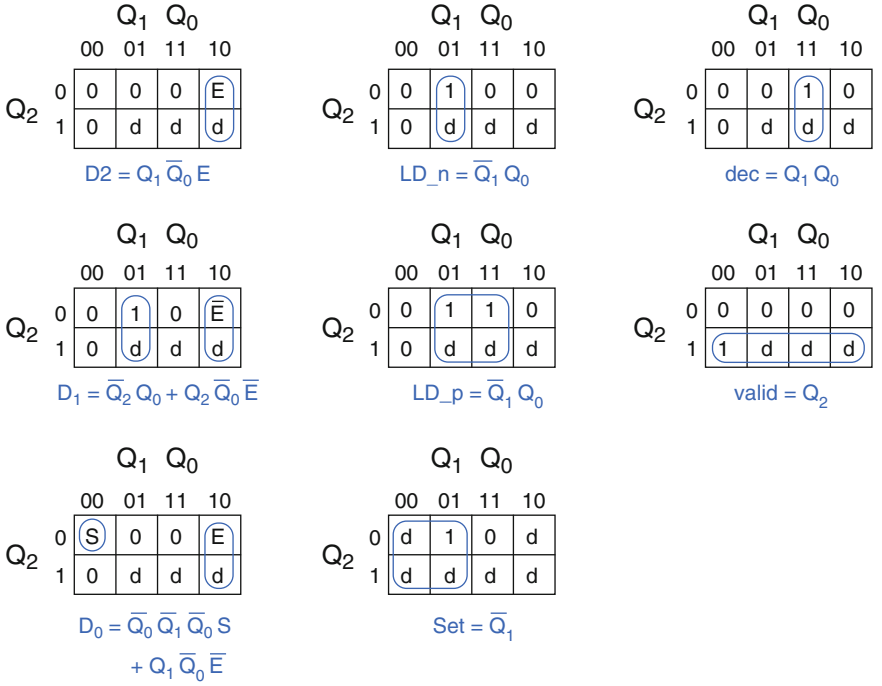


Fig. 18.10 K-maps for the next state and output logic for the datapath controller specified in Table 18.4

Example: Fibonacci Sequence

For our final example, we'll complete the Fibonacci Sequence device from the Datapath chapter. Recall in Fig. 18.12 the datapath for this algorithm.

For the algorithm, we need to initialize the F_{n-1} and F_{n-2} registers as well as n. Then we need to update the values of F_{n-1} and F_{n-2} with the new values ($F_{n-2} = F_{n-1}$ and $F_{n-2} = F_{n-2} + F_{n-1}$) and decrement n. We then check to see if n is zero. If it is,

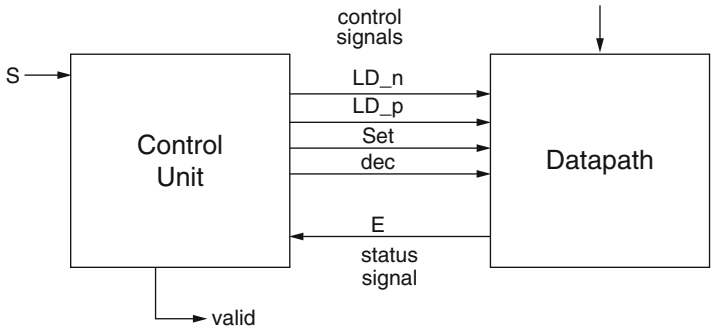


Fig. 18.11 Datapath controller block diagram for the factorial computation system

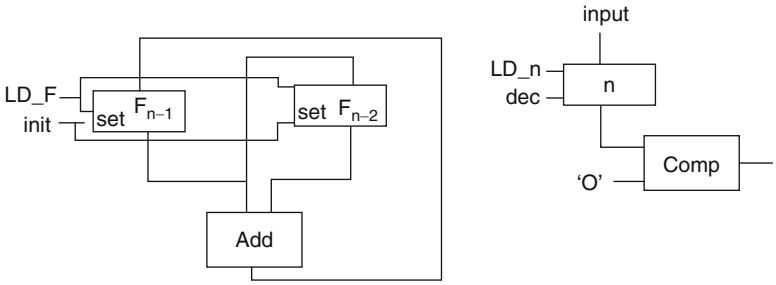
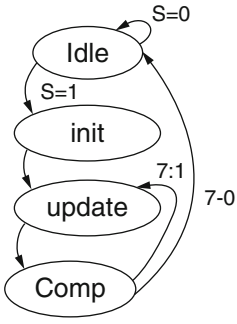


Fig. 18.12 Fibonacci Sequence datapath

Fig. 18.13 State diagram for the Fibonacci Sequence controller



we stop. Otherwise, we repeat. That's it! That's the algorithm, written out in words rather than code. (Remember, it's perfectly valid to do that and, in fact, our understanding of *computation* in the term *computing science* is exactly that: processing of language.)

By now we can pretty readily construct the state diagram for this controller. The only tricky part is to make sure we put a comparison state between decrementing n and evaluating whether it's zero. In Fig. 18.13, we can see this diagram realized.

Table 18.5 State table for state diagram of Fig. 18.13

	Present state		Inputs		Next state		Outputs			
	Q1	Q0	S	Z	D1	D0	LD_F	LD_n	init	dec
Idle	0	0	0	–	0	0	0	0	0	0
			1		0	1				
init	0	1	–	–	1	0	0	1	1	0
update	1	0	–	–	1	0	1	0	0	1
comp	1	1	–	0	0	0	0	0	0	0
				1	1	0				

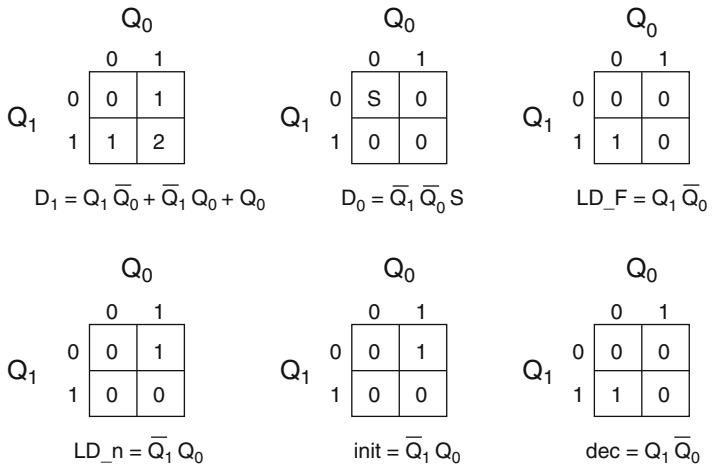


Fig. 18.14 K-maps for the controller state table given in Table 18.5

With only four states, the corresponding state table, seen in Table 18.5, is straightforward to produce.

And, for completeness, in Fig. 18.14 we see the K-maps for the next state and output logic for this controller.

We can see from these equations that there is room for optimization that may not have been obvious without going through all these steps. It's clear the LD_n and init signals can be combined into a single control signal. Good synthesis tools will find and implement this optimization even if the human designer doesn't see it.

Exercises

18.1 Consider the datapath of Fig. 18.15 and this associated algorithm:

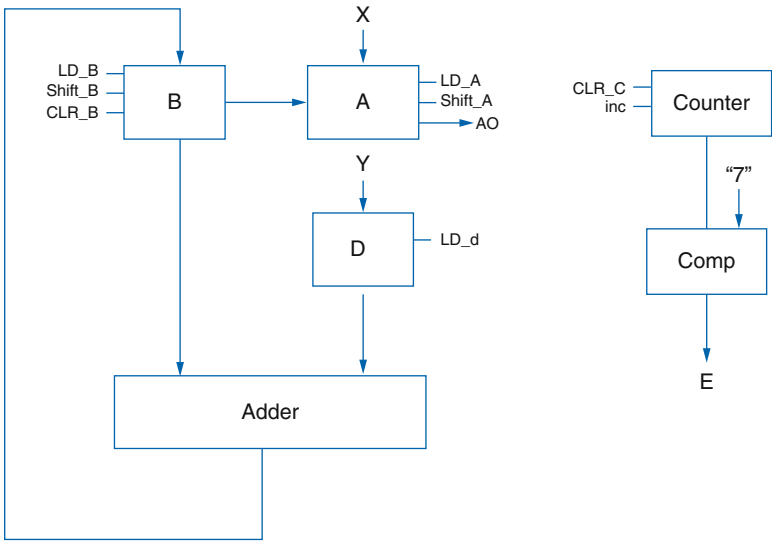


Fig. 18.15 Datapath for exercise 18.1

1. Wait for a start signal *S*
2. Load registers A and D, clear register B, and clear the Counter
3. If status signal A0 = 1, then add B and D and store the result back in B
4. Shift registers B and A and increment the Counter
5. If the status signal E = 1, then return to step 1
6. If the status signal E = 0, then repeat starting at step 3

Diagram the Moore state machine which can implement the given algorithm on the given datapath. Label your states S_1, S_2, S_3, etc. according to which step of the algorithm they are implementing. The inputs to the state machine should be the signals S, A0, and E and the outputs should be the Control Word for the given state listing the control signals in the following order: LD_B, Shift_B, CLR_B, LD_A, Shift_A, CLR_C, inc.

What does this algorithm do?

18.2 Design a controller-datapath system which implements the following algorithm. Draw the datapath and state diagram, fill in the state table, calculate the minimal sum logic for the next state equations, and implement the outputs on a decoder.

```
input n
sum = 0
while (n > 0){
     sum = sum + n
     n = n - 1
}
assert valid and reset
```

18.3 Design a controller-datapath system with three registers A, B, and C that performs the following operations in the order specified:

(a) Load two signed numbers into A and B
(b) If the number in A is positive, divide the number in A by 2 and move the result to C
(c) If the number in A is negative, multiply the number in B by 4 and move the result to C
(d) If the number in A is zero, clear register C

Draw the datapath and state diagram, fill in the state table, calculate the minimal sum logic for the next state equations, and implement the outputs on a decoder. Do not use a multiplier or divider unit; instead use shift registers.

18.4 Design a controller-datapath system for a soda dispenser. The dispenser should initialize and then wait on coins to be deposited and then update a register *total* with the value of the coin before returning to the wait state. Assume there are three inputs to the controller for (N)ickle, (D)ime, and (Q)uarter. If the value of *total* is greater or equal to the cost *S* of a soda, then a *dispense* output is asserted and the system resets itself.

Draw the datapath and state diagram, fill in the state table, calculate minimal sum logic for the next state equations, and implement the outputs on a decoder.

18.5 Design a controller-datapath system that will wait on a start signal S and then sum positive inputs from a keypad until the user inputs a 0. The system inputs are a number X and a signal *(E)nter* from an active-high enter button which indicates the number is ready. The outputs should be the sum of all positive inputs and a valid signal.

Draw the datapath and state machine, fill in the state table, calculate minimal sum logic for the next state equations, and implement the outputs on a decoder.

18.6 Design a controller-datapath system that will calculate the factorial of an input number. The system should take in input number X as well as a start bit S. The system should wait on the S = 1 input, then take in X and perform the calculation. A valid bit should be set when the calculation is complete.

18.7 Design a controller-datapath system that will implement the following algorithm:

```
input n
x = 0, y = 1
while ( n > 0){
    output y
    t = y
    y = y + x
    x = t
    n = n - 1
}
```

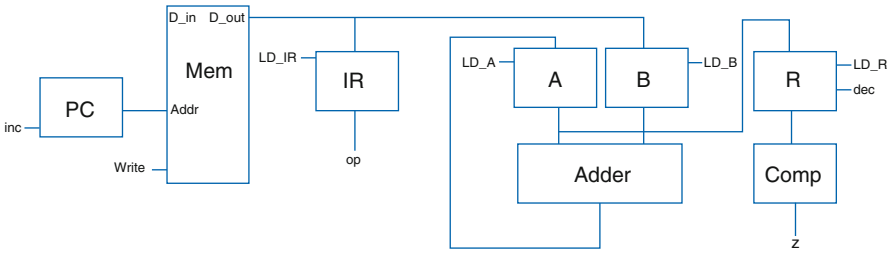


Fig. 18.16 Datapath for exercise 18.8

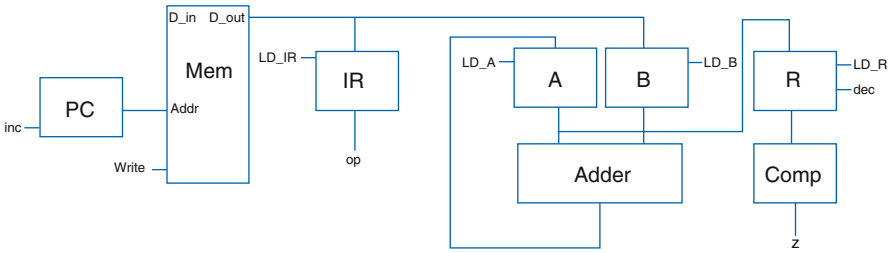


Fig. 18.17 Datapath for exercise 18.9

Draw the datapath and state machine, fill in the state table, calculate minimal sum logic for the next state equations, and implement the outputs on a decoder.

What does this system do?

18.8 Consider the datapath given in Fig. 18.16.

Consider the following algorithm:

1. Load IR from the address stored in PC and increment the PC
2. If op =00, then

   2a. Load B from the address stored in PC and increment the PC
   2b. Add A = A + B

3. If op =01, then move A to R
4. If op =10, then decrement R
5. If op =11, then add A = A + B only if Z = 1
6. Repeat from step 1

Design the state machine controller for implementing this algorithm using this datapath:

18.9 Consider the datapath of Fig. 18.17:

Consider the following algorithm:

1. Load IR from the address stored in PC and increment the PC
2. If op =00, then add A + B and store the result back in A

Table 18.6 State table for exercise 18.10

	Present state		Input		Next state		Outputs					
	Q1	Q0	c_1	c_0	D1	D0	inc	LD_IR	LD_A	LD_B	X_in	Y_in
Fetch	0	0	x	x	0	1	1	1	0	x	0	x
Decode	0	1	x	x	1	0	0	0	0	x	0	x
Execute	1	0	0	0	0	0	0	0	0	1	0	1
			0	1	0	0	0	0	0	0	1	x
			1	0	0	0	0	0	1	0	0	0
			1	1	0	0	0	0	1	0	x	1
	1	1	x	x	x	x	x	x	x	x	x	x

3. If op =01, then decrement R and if, after the decrement, Z = 1, move the value in A to R
4. If op =10, then

   4a. Load B from the address stored in PC and increment the PC twice
   4b. If Z = 1, then add A + B and store the result in A

5. If op =11, then move A to R
6. Repeat from step 1

Design the **Moore** state machine controller which will implement the algorithm using the given datapath. Use the Control Word table below to indicate what the outputs are for each state. Have one row in this table for each state in your controller design.

18.10 Consider the state table given in Table 18.6 for a control unit.

(a) Calculate the minimal sum logic for **D1** and **X_in**.
(b) Implement **D1** on a 4:1 MUX.
(c) Implement **D0** on a 16:1 MUX.
(d) Implement inc, LD_A, and X_in on a single 4:16 decoder.

18.11 Design a controller-datapath system which reads in four temperatures from a digital sensor and outputs C = 1 if the average temperature is equal to or greater than a threshold value V. It should then reset to read in four more temperatures. The inputs to the system are a start signal S and temperatures T. The outputs are the average temperature and C. Note that the temperatures are to be read in one at a time.

18.12 Design the datapath, controller state diagram, and the state table for this device. Also briefly describe how your system works.

18.13 Design a reaction timing system. Upon receiving a start signal S, the system will wait for 10 sec and then turn on a light. The system will record how much time (in ms) elapses between the light going on and the user pressing a button in response. If the light is on for 2 sec without the button being pressed

then the system will time out, set the reaction time to 2000, and assert a slow response output W.

The inputs to the system are the start signal S, the button input B, and a 1 kHz clock input C.

The outputs are the light enable L, the reaction time T, and the slow response W.

Design the datapath, controller state diagram, and the state table for this device. Also briefly describe how your system works.

18.14 Design a datapath that can Fetch, Decode, and Execute the instructions given below. The `JMP` instructions copy the appropriate value (either N or A) into the PC.

Instruction	*Opcode*
MOV A, #N	**000**
MOV Rn, #N	**001**
MOV A, Rn	**010**
MOV Rn, A	**011**
ADD A, #N	**100**
ADD A, Rn	**101**
JMP N	**110**
JMP A	**111**

Fill in the state table for the control unit for your design. Be sure to write down your control signals in the table and label your states.

18.15 Design a controller for the datapath shown in Fig. 18.18 that will (1) wait for a start signal S, (2) read in N, (3) read in A and B, (4) add A + B and then if A > B store the sum in B, otherwise store in C and output the two least

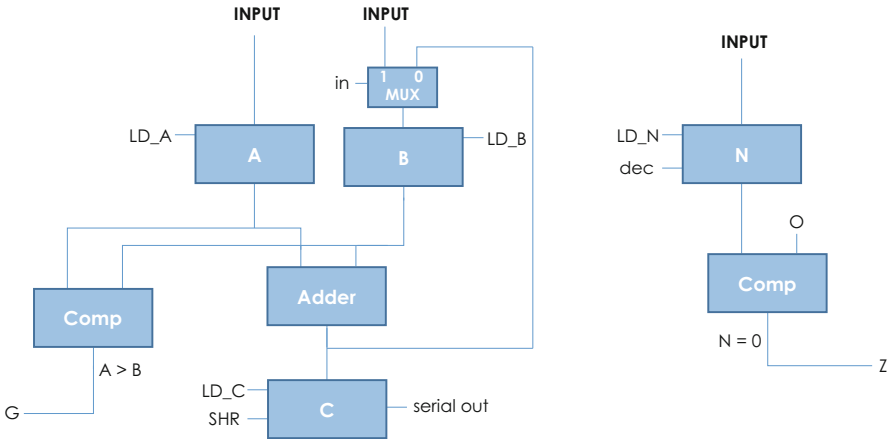


Fig. 18.18 Datapath for exercise 18.15

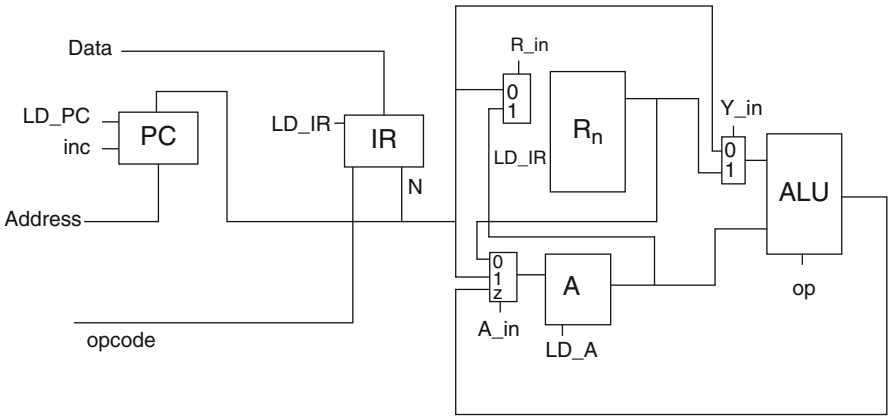


Fig. 18.19 Datapath for exercise 18.17

significant bits of C serially, (5) decrement the count, and (6) if the count is not zero repeat from step 3, else return to step 1.

Draw a Moore machine state diagram for your controller. The inputs to the controller are S, G, and Z.

18.16 Design the state machine controller for the datapath from exercise 15.21.

18.17 Consider the datapath shown in Fig. 18.19.

Instruction	**opcode**
`MOV A, #N`	`000`
`MOV Rn, #N`	`001`
`MOV A, Rn`	`010`
`MOV Rn, A`	`011`
`ADD A, Rn`	`100`
`ADD A, #N`	`101`
`SUB A, Rn`	`110`
`JMP N`	`111`

Remember, `JMP N` moves the value N from the instruction encoding into the PC.

This datapath is capable of Fetching, Decoding, and Executing the eight instructions. The control unit for this datapath is a state machine which will generate the appropriate control signals during each stage of the instruction cycle.

18.18 Design a processor (both control unit and datapath) that can Fetch, Decode, and Execute the given instruction set.

(a) Draw the datapath, being sure to label your control signals.
(b) Next, fill in the state table for the control unit.
(c) Finally, use MUXes to implement the next state equations and use a decoder to implement the output equations.

Instruction	opcode
MOV A, #N	00
MOV Rn, A	01
ADD A, Rn	10
ADD Rn, #N	11

18.19 Design a processor (both control unit and datapath) that can Fetch, Decode, and Execute the given instruction set. The two jump instructions, `JMP A` and `JMP B`, will move the value of the respective register into the PC. For example, `JMP A` will set the PC equal to the value in register A.

(a) Draw the datapath, being sure to label your control signals.
(b) Next, fill in the state table for the control unit.

Instruction	opcode
MOV A, #N	000
MOV B, A	001
MOV A, B	010
ADD A, #N	011
ADD A, B	100
ADD B, #N	101
JMP A	110
JMP B	111

18.20 A very simple approach to multiplication involves repeatedly adding the multiplicand the number of times indicated by the multiplier. For example, in decimal $4 \times 3 = 4 + 4 + 4 = 12$. The following diagram shows an architecture to achieve binary multiplication using this procedure. It consists of a counter, a parallel adder, and two registers. Register A stores the product and, hence, its length is twice the length of register B, which stores the multiplicand. When a start signal S, which is synchronized with the clock, is set to 1, an INIT $=1$ signal is produced by the controller. When INIT $=1$, register A is cleared, the multiplier is loaded into the counter, and the multiplicand is loaded into register B. The multiplicand is then added to register A, as a result of an ADD $=1$ control signal, and the counter is decremented, as a result of a DECREM $=1$ control signal, until the multiplication is completed, which occurs when the counter gets to zero. The controller then asserts the completion signal COMPLETE for one state time, after which it returns to the start state and waits for a new start signal. Construct a state machine controller for this algorithm.

18.21 One way to calculate the two's complement of a number is the following: working from right to left, retain all least significant 0's and the first 1, after which each of the remaining higher significant digits is complemented. The following diagram shows an architecture, along with a summary of the control signals, for a two's complementer based on this algorithm consisting of an 8-bit shift register and a mod-8 binary counter. When a start signal S is

set to 1, the register is parallel loaded with an 8-bit binary number and the counter is reset to 0. The two's complementing algorithm is then carried out. The least significant bit of the register is checked and the appropriate bit of the complemented number is determined and entered into the high end of the shift register upon a shift-right command. The process is continued until the two's complement of the original number appears in the shift register. Upon each shift, the counter is incremented to provide an indication of when the process is complete. After forming the two's complement, the system returns to its initial state to await another start signal. Construct a stat machine for the controller.

18.22 The datapath in Fig. 18.20 can perform the restoring division algorithm:

1. Put dividend in lower half of the dividend register and clear the upper half. Put divisor in divisor register. Initialize quotient bit counter to zero.
2. Shift dividend register left 1 bit.
3. Load the difference of the high byte of the dividend and the divisor into the upper half of the dividend register.
4. If the difference is negative, then add the divisor to the high byte of the dividend and store back into the dividend high register and shift a 0 into the quotient.
5. If the difference is positive, then shift a 1 into the quotient.
6. If the quotient bit counter is not yet at n, then repeat from step 2.

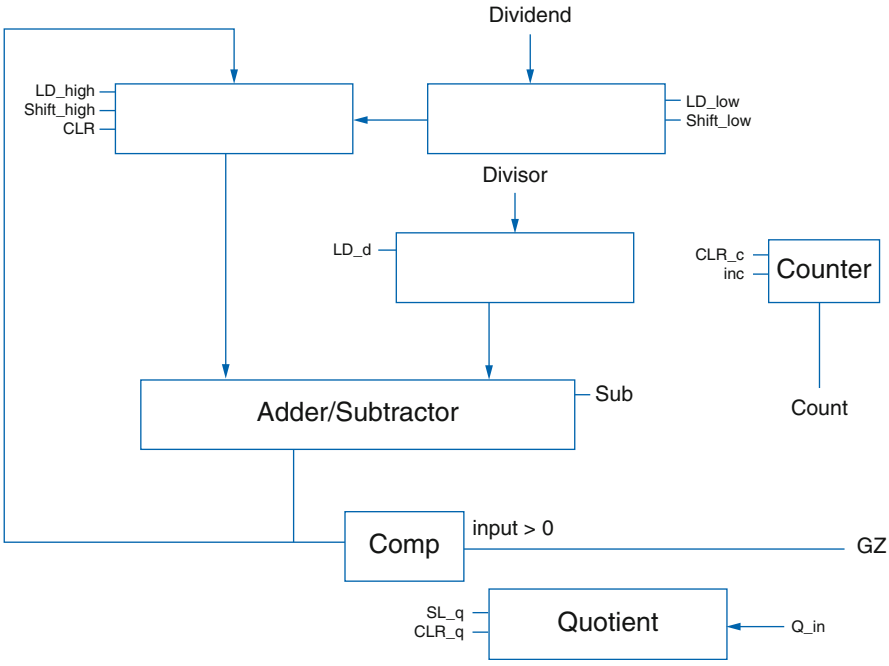


Fig. 18.20 Datapath for exercise 18.22

The inputs to the controller are a start bit *s*, the current *count*, and the comparator output GZ. The outputs of the controller are the control signals listed for the components, the Q_in input to the quotient register, and a *complete* signal when the calculations are finished.

Design the controller for this datapath.

18.23 Consider the following instruction set:

`LD A`	`011xaaaa`	Load A with the contents of memory location `aaaa`
`LD B`	`100xaaaa`	Load B with the contents of memory location `aaaa`
`ADD A, B`	`101xxxxx`	A ← A + B
`DIV AB`	`110xxxxx`	A ← quotient, B←remainder
`JNZ addr`	`111xaaaa`	If A is not zero, then jump to address `aaaa`

(a) Design a general purpose datapath which will implement these instructions. Clearly label your control and status signals and identify your components and their features.

(b) Draw the state machine diagram for a controller for this datapath, derive the state table, and solve for the next state and output logic.

18.24 Consider the datapath given in Fig. 18.21 and the associated instruction set.

Instruction	**opcode**
`MOV A, #N`	`00`
`MOV B, A`	`01`
`ADD A, B`	`10`
`ADD A, #N`	`11`

This datapath is capable of Fetching, Decoding, and Executing the four instructions. The control unit for this datapath is a state machine which will generate the appropriate control signals during each stage of the instruction cycle.

Build the control unit for this datapath.

18.25 Suppose we have a ROM unit with 256 stored 8-bit values. We want to take an input B and output the number of times B occurs in the 256 entries in the ROM unit. As usual, we wait on a Start signal S to begin and output a valid signal V when finished. Partition the design into a controller and a datapath. Show the block diagram for the controller, clearly labeling your components and control signals. Show the state diagram for the controller and give a table showing what output signals are asserted in each state.

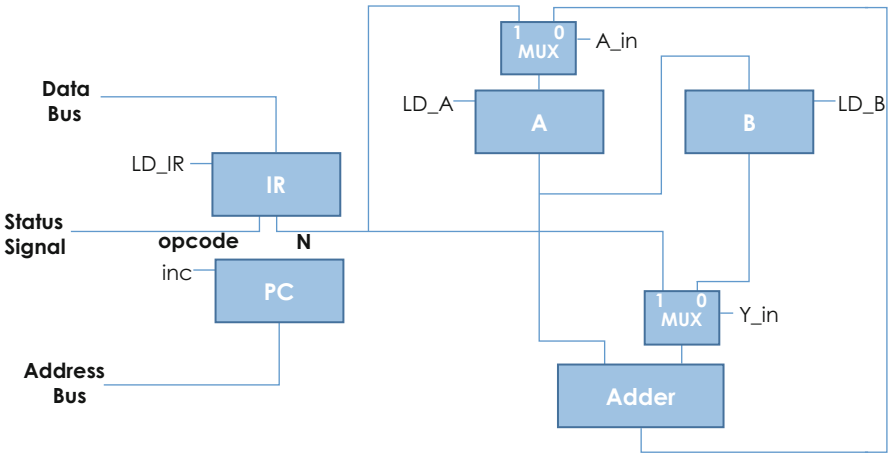


Fig. 18.21 Datapath for exercise 18.24

18.26 Consider the following algorithm:

```
sum = 0;
t = 0;
while (t < 256){
sum = sum + abs(a[t] - b[t]);
t = t + 1;
}
```

In this algorithm abs is the absolute value and a[t], b[t] represent inputs a and b at time t.

Design a device, partitioned into controller and datapath, that can implement this algorithm.

The datapath should have input lines a and b (but you do not need registers for these—assume the inputs are always retrieved from an external source once per clock cycle), storage for sum and t, and a way to run through the while loop.

Draw a state diagram for a controller that will implement the algorithm on your datapath. Also give the state table and compute the next state and output equations.

18.27 Design a datapath-controller system which will input two unsigned numbers and output either (1) the average of the two numbers if the largest number is greater than 100 or (2) the smallest of the two numbers if the largest number less than or equal to 100.

Take into account the following design constraints:

- Wait to begin the process until a start signal S = 1 is detected
- Have only a single input data bus (so that both numbers must be input on the same bus on successive clock cycles)

- Output a valid bit V when the final result is determined

Draw the datapath and give the state machine controller for this system.

18.28 Design a controller-datapath system which can input an array of 256 8-bit values and calculate the maximum difference between any two elements in the array. That is, the system should read in the entire array, calculate the largest value in the array, calculate the smallest value in the array, and then output the difference.

You may assume inputs are unsigned and use a single register bank with 256 8-bit registers.

You should wait for a start signal s before beginning and the algorithm should return to this waiting state when the difference has been output. Use a valid bit to determine when the output is ready.

Chapter 19
State Machine Theory and Optimization

In Chap. 17 we learned how to design, use, and implement state machines. In Chap. 18, we employed these machines as controllers for datapaths. While doing this, we saw their utility in tracking the steps of an algorithm. As a state represents the most important information about a given system and, due to our timing rules, we understand the changes made as we transition from state to state, it's natural to use a state in this manner. But, as we may recall from all the way back in Chap. 1, we actually understand an algorithm to be an expression of language. Sure, it's a computer language which is (provably) less expressive in key ways than human languages, but yes, indeed, it is a language. In this way we can concept the state machine as a language processor.

In fact, the way state machines were used when humanity first started pondering algorithms as abstract realities worthy of study and reflection was in this same way! The only difference is that they didn't have CMOS transistor arrays and logic gates in which to realize these state machines physically. Instead, state machines existed only as theoretical constructs that were *models of computation*. That is, according to Alan Turing and others, state machines were used as stand-ins for the digital electronic computers that weren't yet in existence. They were able to study algorithms and some aspects of computing itself through the use of state machines. While engineers today consider these as sequential logic circuits, the theoretical properties of state machines transcend any particular implementation in a given technology.

This chapter has something for everyone! We briefly introduce a mathematical approach to language and processing using state machines as models of computation. We also discuss the state assignment and state reduction problems which have significant relevance to engineering circuit-level design. In fact, these are so important they are still open areas of software research within synthesis tools which try to produce the best transistor implementation of a specified state machine.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_19

Formal Languages

Since state machines began their life as tools for processing languages, it's important first to investigate what computer scientists call *formal languages*. Now, this is an inexhaustible topic full of intricacy and depth. One could spend an entire career producing scholarship in this area. Rest assured our development here is in accord with the rest of this text. While remaining rigorous, we'll cut to the chase.

We can define an *alphabet* A as simply being a set of characters and a *string* S over A as being zero or more elements in a given order. Then a *language* L is just a set of strings.

For example, in this course we are limited pretty much to the alphabet A = {0,1} and all our strings are ordered sequence of these 1's and 0's. The languages we have studied have been made up of sequences of these strings. In Chap. 10 we see how to represent code as strings over the alphabet {0,1} and so in this way we see the mathematical definition here matches our understanding because certainly computer programming languages ought to be included within the definition of a formal language defined herein.

More generally, though, we can consider larger alphabets. Take the entire English alphabet, for example, consisting of the letters a through z. In this case, the strings are what we call words and the language is the set of all combinations of words. Everything from this textbook to Shakespeare to Dante (in translation) to the assembly instructions for your desk chair are parts of this language. Most languages studied fall between these extremes.

Of interest in computing are languages with certain structures to them. The *regular languages* are those which are closed under the operations of addition and multiplication (among a few other things.) This means if you take two strings in a regular language and you add them or multiply them then the resulting string will also be in the language. We can think of addition and multiplication here as understood within the paradigm of Boolean Algebra as OR and AND operations.

In a process that ought to remind us of the sequence detectors of Chap. 17, we can use state machines to *recognize* regular languages. The idea is that computation itself, as a process worthy of meta-analysis, is exactly that: the character-by-character processing of languages. To the extent, then, that we can say interesting things philosophically or mathematically about this process, we have made a scholarly contribution. This is why the state machine is said to model computation. It is within a framework similar to this that the original theoretical computer scientists—most of them trained as mathematical logicians well versed in the work of George Boole before a single electronics engineer had ever heard of him—developed the core ideas of computation that would be developed later into real working machines.

So, let's look at an example. Let's build a state machine to recognize a particular set of regular languages over the alphabet {0,1} that we are familiar with. What this means is that we can receive, one by one, 0 or 1 inputs in any combination of ORs

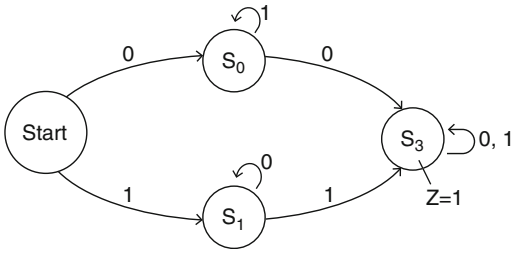


Fig. 19.1 A state machine that recognizes languages over {0,1}

and ANDs. Consider the state diagram of Fig. 19.1 and we'll discuss its components.

This state machine is said to *recognize*, or successfully process, languages that cause the state transitions to end up where we'd expect: in the state that asserts the output signal z. So, for example, in the input sequences 011 and 1000 both end in states S_0 and S_1, respectively. Since they do not end in the state S_3 where $z = 1$, they are not accepted by the state machine and are therefore not part of the language this machine recognizes. The strings 011100 and 11111, in contrast, and indeed in the language the machine recognizes because they both end up in S_3 with $z = 1$ output. In fact, we can put on our mathematician hats a bit here and see that all strings of the form 01^n0 or 10^n1 are accepted. In the most general form we'd write this as 01*0 (0 + 1)* + 10*1(0 + 1)*.

Well, that's certainly cryptic. Let's unpack it. Remember, as in our Boolean algebra discussions, + is OR and multiplication is AND. So, the first thing we notice is that this expression is made up of the two terms 01*0(0 + 1)* and 10*1(0 + 1)*. The first term starts with a 0 and indicates the transition from the Start state to S_0 while the second term starts with a 1 and indicates the transition from the Start state to S_1. Then, the first term, representing the state machine in state S_1, has a 1* expression. The * notation means that we can receive any number of 1's. This is indicated in the diagram with the transition arrow from S_1 back to S_1 when the input is 1. We stay in that state as long as we keep receiving 1's. A similar thing is happening in S_2 with the 0* expression. We can receive any number of 0's after receiving the first 1 and transitioning from Start to S_2. Those are then followed up with a 1 or a 0 which can be seen in the diagram as the transitions from S_1 and S_0 to S_3. Each term ends with the expression (0 + 1)* which can be understood as receiving any number of 0's or any number of 1's (combining the + and * notations here). We observe this in the diagram transitions from S_3 back to S_3 on receipt of either 0 or 1.

Wow, yeah, so welcome to theoretical computer science! It gets better from here if you can imagine that! The point of this is not to get caught up in mathematical notation but to see the deep connection between our sequence detector state machines and the canonical use for these state machines in the study of models of computation at the highest levels.

One thing that the more mathematically inclined may be asking right now is "What if you get an infinite number of 1's or 0's while in either of states S_1 or S_0? If

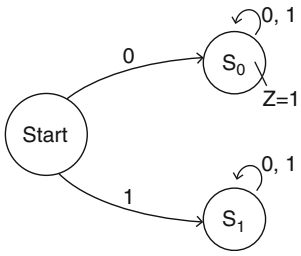


Fig. 19.2 Finite automaton that recognizes languages over {0,1}

the * notation means we can read in any number of 1's or 0's then what if the input string simply never ends? Would that be part of the language?" This is actually a great question and we answer it by using the time honored technique of sidestepping it completely. Left unsaid thus far (and therefore open game for the mathematically precise) is that strings are *finite*. Languages themselves are *infinite* by nature, but the individual strings making up a language are themselves finite. In fact, computer scientists call these *finite state machines* or, more impressively, *finite automata* because, well you know, *automata* sounds more technical than *state machine*. (But really they are the same thing. The engineering community tends to favor *state machine* while computer scientists say *automata*.) While infinite state machines are worthy of study, our entire discussion in this text relates to the finite. So when we say * permits any number of 0's or 1's, we mean it in the sense that we must pick an actual number and repeat that many times *and no more*.

Before moving on, let's look at one more example of this. Consider the state diagram in Fig. 19.2.

This machine will recognize all strings of the form 0(0 + 1)*. Make sure you can see that from the diagram and understand the use here of the + and * notation. It's very helpful (and technically precise) to relate these operations to the diagrams themselves. The + represents options in transitions while the multiplication represents dependent transitions. We *must* receive a 0 to start off in order to get to the only state that asserts an output z = 1. Therefore, we must have an input string of 0 AND something else, with that something else being any combination of 0's and 1's that follow. This is written as in Boolean algebra as 0(0 + 1)*. Seriously, the picture is NOT a step down from the mathematical notation. In fact, it is an important result of theoretical computer science that the set of all regular languages matches exactly the finite automata. So, we can understand all these languages through the diagrams.

The diagrams of Figs. 19.1 and 19.2 are actually called deterministic finite automata. We can also speak of nondeterministic finite automata, but these are both impossible to realize physically (so of less inherent interest to us) and exactly as powerful in terms of the languages they can recognize as the deterministic finite automata. So, while they form an important part of a full study of automata theory, they are too far off course here to get into.

It turns out that this is just the beginning of the study of state machines. Regular languages are useful (and the associated *regular expressions* are constant

companions of programmers of certain stripes) but they are not expressive enough to capture the intricacies of many programming languages we want to study. After all, if we want our computers to recognize code, we'd better be able to diagram the reception of something like COBOL or Pascal or C++. And these languages have features that cannot be modeled via regular languages. (A full study of formal languages gets into these details in much more depth if the interested reader is intrigued by this idea and wants to pursue the topic further.) We have to study *context-free languages* instead if we want to build models of computation capable of processing these more advanced coding approaches. What is a context-free language? That's a great question for another text. Our goal here was simply to introduce the idea of formal languages and relate them to state machines and see where these incredibly important concepts had their genesis. In some sense, when we are building our simple sequence detectors we are covering the same ground as the early computer scientists as they tried to form the basis for the study of computation and algorithms as a separate field of human knowledge. In our case, however, we have transistors and are able to actually implement these things in physical devices. Both sides—the philosophy and the engineering—are vital to progress in these areas, and hopefully this brief section on the topic serves as a motivating introduction to this enlightening and awesome pursuit.

State Reduction

No matter what our end goal, be it to study computation itself or to build powerful and efficient electrical circuits, we have an interest in efficient state machine implementations. We observed in an ad hoc fashion in Chap. 18 that we can, as human designers, find efficiencies regarding the number of states in a system. Through careful consideration of the needs of the algorithm and the steps already encoded in the machine, we were often able to reduce the full state machine to one with fewer states. This saves us in terms of execution time because, since each state transition occurs on a clock cycle edge, if we have fewer states to traverse then we can, at least in theory, run through the algorithm the machine is driving in a lesser amount of real time. This makes our computations faster and more efficient and that's a great thing all around. Another benefit to reducing the number of states is that we can potentially use fewer bits to represent the state (in the event we reduce past a power of two, such as from 9 to 7 states) or at least hope that the next state or output logic requires fewer gates as a result. In the end, following the basic principle of engineering that says we want to build efficiently, we have interest in the problem of *state reduction*.

But, we don't want to change the behavior of the state machine! We want to reduce in such a way that the machine retains its functionality. That is, whatever sequence of inputs is sent into the machine must produce the same sequence of outputs in the reduced state machine as it did in the original unreduced machine. To

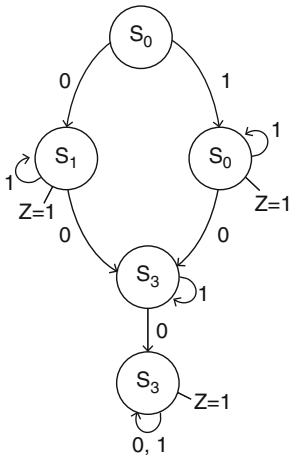


Fig. 19.3 State machine with reducible states

ensure this, we need to decide what it means for states to be *equivalent* and then frame our reduction process in terms of *eliminating equivalent states*.

So, definition time: two states are said to be equivalent if they have the same next states and outputs for each set of inputs. That is, two states are "the same" if the transitions out of the states all produce the same overall effect on the system.

Consider the state machine given in Fig. 19.3.

Let's think through things and try to find out which states might be equivalent. First, we can list the states based on output, as two states that produce different output cannot possibly be equivalent. This gives us *partitions* $\{S_1, S_2, S_4\}$ and $\{S_0, S_3\}$. It's impossible for states in one partition to be equivalent to states in another partition. So our thought process is now to look within partitions for equivalencies. If we study the diagram, we can start to get an intuition of the fact that we're moving towards collapsing S1 and S2 into a single state. It might at first glance appear they cannot be equivalent because they do not share exactly the same next states. Sure, for input 0 they both transition to S_3, but what about for input 1? Well, in that case they both transition back to themselves. But, this is circular, because if they were the same state then the transitions for input 1 would indeed go back to the same state, call it S_{12}.

What we are doing is replacing the next state as an individual state with the next state as element of an equivalency class. That is, once we partition based on the outputs, we can then compare next states based on equivalency classes rather than specific states. If two states in a partition coincide on which partitions their next states are in, then they can be reduced into a single state (Fig. 19.4).

So, we can reduce one of the states S_1 or S_2 and end up with the reduced state machine of Fig. 19.3.

Now let's look at the more complex example given in Fig. 19.5.

This diagram represents a Mealy machine so when we partition based on outputs we must keep in mind that these outputs are associated with inputs and not just based on the states themselves. As we only have two states with non-zero outputs, the first

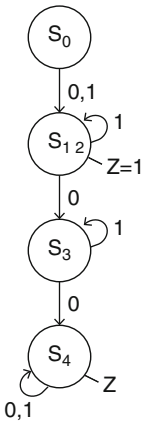


Fig. 19.4 Reduced state machine of Fig. 19.3

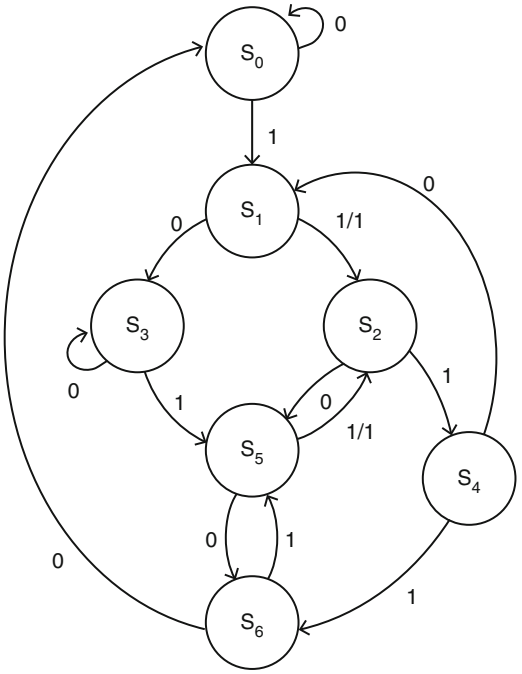


Fig. 19.5 Reducible Mealy machine

pass at partitioning is straightforward: {S0, S2, S3, S4, S6} and {S1, S5}. From there we further partition based on equivalency classes of next states. We can see that S0, S3, and S6 all transition to the class {S1, S5} on an input of 1 and back to their own class on an input of 0 while states S2 and S4 show the opposite behavior. Both S1 and S5 share output classes and thus remain in the same partition. Therefore, our next pass at partitioning gives us {S0, S3, S6}, {S2, S4}, and {S1, S5}.

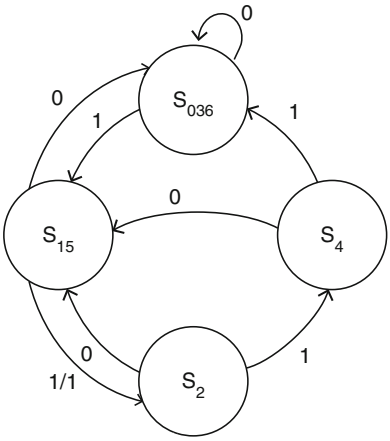


Fig. 19.6 Reduced state diagram of Fig. 19.5

Now it's a bit tricky: notice how when we analyzed S2 and S4 in the last step we concluded their next states were to the same equivalency classes? But, when we update the contents of our partitions, we must check all of these next states again to see if it's still the case that the next states line up appropriately. Thus, we end up with an interative process to reduce the states.

In this case, an examination of the states in the first class {S0, S3, S6} reveals that yes, they do still all transition on a 1 to members of partition {S1, S5} and on a 0 to partition {S0, S3, S6}. So no changes are required here. The same is true for {S1, S5}. For {S2, S4}, however, we have a new development. On an input of 1, S2 remains within {S2, S4} while S4 transitions to {S0, S3, S6}. Therefore, we must split S2 and S4 into different partitions.

This process yields the following state equivalency classes: {S0, S3, S6}, {S2}, {S4}, {S1, S5}. Another iteration reveals no need to further break apart the partitions. Thus, we've discovered the reduced form of the state machine. The diagram is given in Fig. 19.6.

This is a really instructive process. While the first example was something that could potentially be deducible from the diagram and some thought, this one is way more challenging for most to analyze. The two iterations, one to partition {S0, S2, S3, S4, S6} and the next to split it into {S0, S3, S6}, {S2}, and {S4} is not a trivial mental computation. This is why an organized approach is needed and writing out all the steps following the basic rules of first partition based on outputs and then partition based on equivalency classes of next states, is important. The programmers among the readership may be whetting their whistles at the thought of automating this, as it's not hard to see this as a while loop comparing elements in various linked structures based on data from a state table stored in some sort of array. In fact, algorithms close to this are indeed implemented in professional synthesis tools to help optimize the state machines specified by human designers. It's interesting to note that we haven't yet come up with a provably perfect technique or one that scales all that well (consider the number of state comparisons

required in this algorithm.) There is still room for advances in this area of research, so the algorithmically inclined among you may be inspired to contribute to an area of deep theoretical as well as practical interest.

State Assignment

The idea of state reduction is of use to both pure computer scientists and applied engineers. In this section we'll look at an optimization technique that is tied directly to physical implementations of these machines in hardware. Leaving the abstract and going to the other end of the spectrum, we arrive at the engineering need to minimize the underlying hardware implementation of the state machines in ways beyond that potentially realized via the state reduction process.

This section deals with the *state assignment* problem. In Chaps. 17 and 18 we simply labeled the states in their normal binary order and we've made the claim that, in fact, the 1's and 0's encodings we associate with each state is arbitrary and has no effect on the logical operation of the device. That is true, but what was left unexplored was the way the assignment of encodings to states can affect the forms of next state and output equations for a given state machine. It turns out that some state assignments lead to simpler forms of these equations than others.

From the start it's worth stating that, as in state reduction, the state assignment problem is not completely solved. We have some general guidelines to follow regarding what makes a particular assignment better or worse, but there is no known algorithm that can, for all state machines, guarantee a minimal assignment (well, at least in a computationally reasonable amount of time as the obvious brute force algorithm of simply trying every possible combination is order O(n!) is impossible to rely on for general work.) This is exciting, as it means there is still more left within the field to explore! It's also a bit frustrating, because it means our process here, if we are trying to minimize the logic, must be approached with heuristics.

The best way to explain what's going on with state assignment is by walking through an example. So, let's consider the state machine of Fig. 19.7.

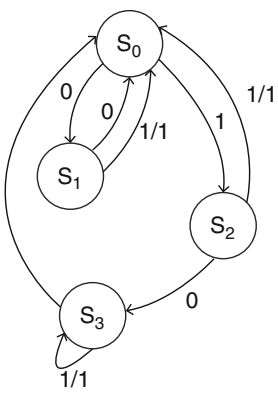


Fig. 19.7 State machine for state assignment optimization

Table 19.1 Binary state encoding for the state machine of Fig. 19.7

	Present state		Input	Next state		output
Q1	Q0	x	D1	D0	z	
S0	0	0	0	0	1	0
			1	1	0	0
S1	0	1	0	0	0	0
			1	0	0	1
S2	1	0	0	1	1	0
			1	0	0	1
S3	1	1	0	0	0	0
			1	1	1	1

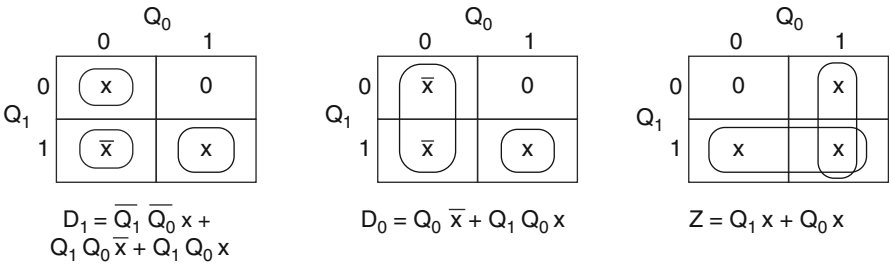


Fig. 19.8 K-maps for the next state and output logic specified in Table 19.1

We see this has four states S_0 through S_3 and, following our development in previous chapters, it's perfectly natural to encode these states in a natural binary order as shown in Table 19.1.

So far, there is nothing new here. The next state and output equations are computed via K-maps as shown in Fig. 19.8.

The D_1 equation, in particular, stands out as cumbersome. Let's think about why that may be. We are aware that scattered 1's in the truth table create terms that don't benefit from grouping and are therefore irreducible. It can be concluded then, that if we can come up with a state assignment that minimizes the number of isolated 1's we would expect to see gains in the resulting logic.

But, how do we accomplish this? What we want to do is to minimize the number of bits that must change from one state transition to another. Ideally we'd like a state and its next state to differ by at most a single bit. When we have to incorporate multiple bit changes in a transition, such as from S_0 to S_3 when $S_0 = 00$ and $S_3 = 11$, we risk more complex logic.

A good rule to try to follow, then, is to try to ensure that states and their next states differ by 1 bit. If we have a reasonably orderly state machine where transitions proceed from left-to-right or from top-to-bottom in a structured way without looking like spaghetti dropped all over the floor (compare, for instance, Figs. 19.1 and 19.5) then we can use a unit-distance representation such as the Gray code discussed in Chap. 9 to number the states.

Table 19.2 More optimal state assignment for the state machine of Fig. 19.7

	Present state		Input	Next state		output
	Q1	Q0	x	D1	D0	z
S0	0	0	0	1	1	0
			1	1	0	0
S3	0	1	0	0	0	0
			1	0	1	1
S2	1	0	0	0	1	0
			1	0	0	1
S1	1	1	0	0	0	0
			1	0	0	1

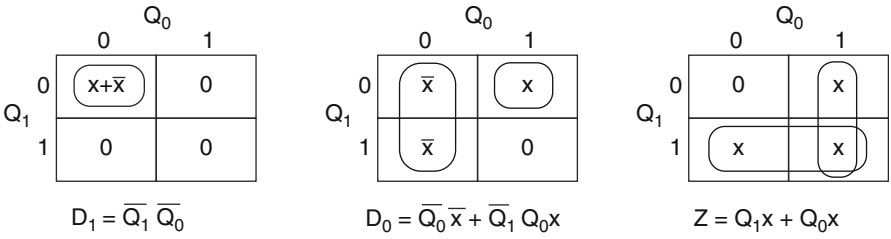


Fig. 19.9 K-maps for the next state and output logic specified in Table 19.2

But there are other considerations. We may see that if we have two states that share a next state that they ought to have a one-bit difference even though they don't transition to each other. Or if two states are themselves both next states of a given state then they should be encoded with a one-bit difference. In reality, there are many "rules of thumbs" proposed and one can only follow so many of them because they end up conflicting with each other. Again, research into the state assignment problem is still ongoing and many heuristic approaches are in use in professional digital design software.

In the case of the state machine of Fig. 19.7, we can try the state assignment in Table 19.2.

The next state and output equations for Table 19.2 are shown in Fig. 19.9.

Compare the next state equation for D1 between the state assignments used in the K-maps in Figs. 19.8 and 19.9. It's clear that the state assignment that incorporated some thought about the process yields better logic. It's not so clear that producing such assignments is a straightforward affair, or that such a result can always be obtained for a particular machine. Still, it's a good takeaway to understand that the encoding we use for the states in our state machines is, as emphasized in Chap. 9 for any digital representation of any sort of information, absolutely within the purview of designers and part of the craft of digital design.

Exercises

19.1 Each of the following regular expressions represents a given regular language. For each one of them, construct the state machine (finite automaton) that can recognize it.

(a) 0 + 1
(b) 01* + 0
(c) (00 + 11)(0 + 1)*
(d) (01)*10(0+1)*
(e) 0 + 1*

19.2 Design a state machine that can recognize strings from the alphabet {0,1} that contain either 00 or 11 as subsets.
19.3 Design a state machine that can recognize strings from the alphabet {0,1} that have 1 as the second to last element.
19.4 Design a state machine that can recognize strings from the alphabet {0,1} that contain neither 00 nor 11.
19.5 Give the language recognized by the state machine of Fig. 19.10.

19.6 Give the language recognized by the state machine of Fig. 19.11.

19.7 Consider the state machine of Fig. 19.12. Reduce the states and draw the new diagram.

19.8 Consider the state machine of Table 19.3. Reduce the states and write both the state table and state diagram of the reduced state machine.

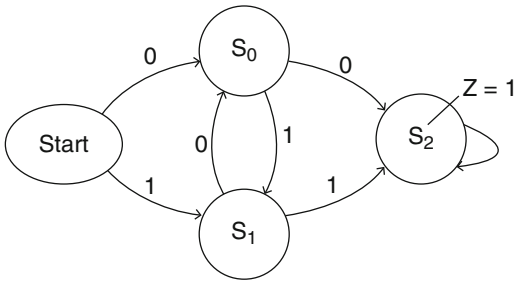


Fig. 19.10 State machine for exercise 19.5

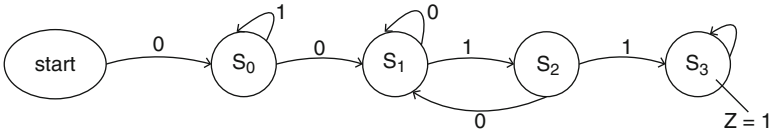


Fig. 19.11 State machine for exercise 19.6

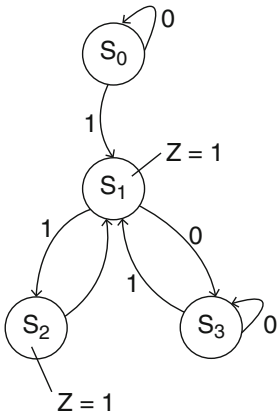


Fig. 19.12 State machine for exercise 19.7

Table 19.3 State table for exercise 19.8

	Present State			Input	Next State			Output
	Q2	Q1	Q0	x	D2	D1	D0	z
S0	0	0	0	0	0	0	1	0
				1	0	1	0	0
S1	0	0	1	0	0	1	0	1
				1	0	1	1	0
S2	0	1	0	0	0	1	0	0
				1	1	0	0	0
S3	0	1	1	0	1	0	1	0
				1	1	0	0	0
S4	1	0	0	0	1	1	0	0
				1	0	0	0	0
S5	1	0	1	0	1	0	1	1
				1	1	1	1	0
S6	1	1	0	0	0	0	0	1
				1	0	1	1	0
S7	1	1	1	0	0	1	1	1
				1	1	0	1	1

19.9 Reduce the number of states in the state machine given in Fig. 19.13. Show the resulting diagram.

19.10 Consider the state machine in Fig. 19.14. Reduce the states and draw the new diagram. What does this machine do? Take care to note that while you draw it up the first time according to a given algorithm, the state reduction process can actually show you some efficiencies to be had in the actual implementation. This may feed back into the algorithm itself at times.

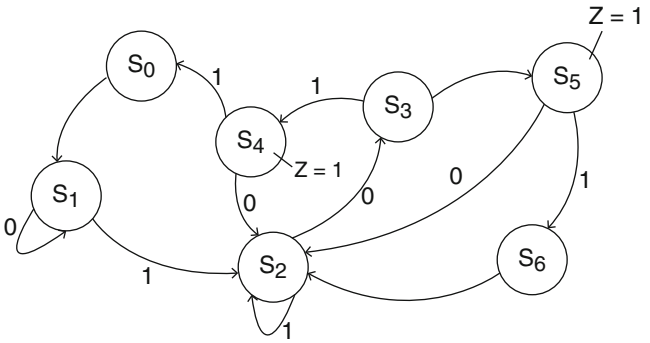
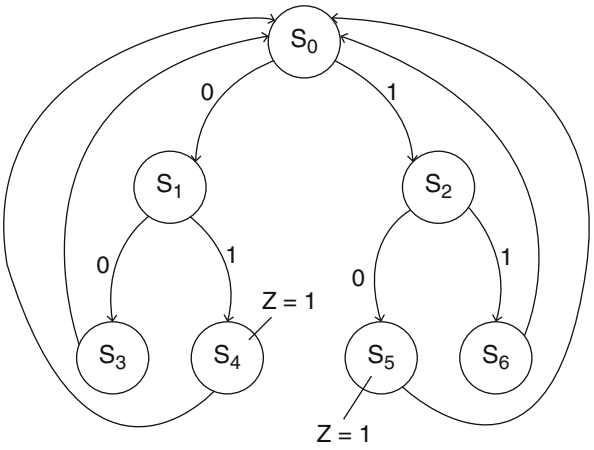


Fig. 19.13 State machine for exercise 19.9

Fig. 19.14 State machine diagram for exercise 19.14.



19.11 Design a state machine to produce an output of 1 for every input of 1 proceeded by three 1's. Here is an example input/output sequence:

x = 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 1 1 1 0

z = 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0

Be sure to reduce the states and draw the diagram of the reduced state machine.

Chapter 20
Instruction Processor Design

In this chapter, we will walk through the entire instruction processor design process from start to finish. This is a culmination of everything we've learned thus far, and this chapter is one long extended example. You should be able to read it now, and recall the chapters in the text where you first encountered the various concepts being described. We will use what we've learned so far to design a computer from scratch. This chapter repeats material from earlier chapters, and seeing it again should further reinforce the concepts and, perhaps most key of all, provide you with the feeling of mastery of using digital logic for computing that is the entire reason for this text's writing. So, above all enjoy, but go through this chapter with the mind of one who is no longer a beginning. "Ah yes, I recall that" should be on your breath as you nod along while finishing the text. The exercises take you through the design described in this chapter.

What Is a Computer?

For our purposes, a *computer* is a device that *executes instructions sequentially*. What does this mean? Let's take these words one at a time:

Instruction A computer can do a lot of different things. Each thing it can do is called an *instruction*. We've designed datapaths that do things like "move the value from register A into one of the registers R0 through R3" and "add the value stored in A to one of the registers R0 through R3 and store the result back in A." These are examples of the types of things computers can do and are, therefore, each an example of an instruction. We write these instructions using shorthand notation such as `MOV R2, A` and `ADD A, R1`. Each computer can do different things, and therefore requires different instructions. When you write a program in a high level language like C++ or Java, a program called a *compiler* will translate your code into instructions like `MOV A, R2` and `SUB A, R1` that the computer is able to process.

J. Seiffertt, *Digital Logic for Computing*, DOI 10.1007/978-3-319-56839-3_20

Execute To *execute* an instruction is to perform the desired operation using an appropriate datapath. We've designed a lot of computer datapaths throughout this text, and each one was capable of executing a different set of instructions. We also know that in order for a datapath to implement an operation (execute an instruction) the appropriate Control Word must be input to the datapath. A state machine controller keeps track of which instruction is being executed and issues the relevant Control Word to the datapath.

Sequentially Instructions are placed in order by the programmer (or compiler) and it is the job of the computer to execute them in this *sequence*. To do this we have to keep track of both the order of the instructions and the instructions themselves. Instructions are encoded as 1's and 0's and stored in memory units just like other data. Since they are in memory units they are accessed via addresses. The order of the instructions, then, is determined by their addresses. A special register in the computer called the Program Counter (PC) keeps track of what address is due to be executed next. Another special register called the Instruction Register (IR) stores the encoded 1's and 0's version of the instruction being executed. In this way—using both of these special registers, the computer can execute instructions sequentially.

A modern *computer system* includes a lot of things besides just the *computer* outlined above: memory, input/output peripherals, timers, co-processors, etc. What we mean by *computer* is really what other people mean by *processor*. This computer, or processor, is an integrated circuit (chip) that executes instructions sequentially. These are made by companies such as Intel and AMD and you usually have one of these in each computer system. When you go to Best Buy and talk to the salespeople about buying a "computer", however, they understand that you want a tower and monitor and keyboard and memory and a mouse and a whole lot of other stuff in addition to just the processor. In this text we have been using *computer* and *processor* interchangeably and the development in this chapter will take you through the design of the processor. How to design and integrate other components of the computer system is covered in other texts in the computer engineering domain.

Design an Instruction Set

The list of things we want the computer to do is called the *instruction set*. The first step in designing a computer is to list all the instructions we want it to execute. Let's think about some good things we want a computer to do and write a list of instructions. We have three main categories of instructions: Data Movement, Arithmetic/Logic, and Control.

Data Movement Instructions

Remember that the computer—the processor—consists of a datapath and a controller. It stores the information it needs in registers in the datapath, but most of the information it will need is stored in memory units located elsewhere in the *computer system*. So, the first thing that the computer needs to be able to do is interact with these memory units to load/read or store/write data.

Let's say our computer has a special register called A and eight general-purpose registers called R0-R7. Then we can have the following instructions:

- Load a value from memory into register A
- Store a value from register A into memory

We could also have the following:

- Load a value from memory into any of registers R0 through R7
- Store a value from any of registers R0 through R7 into memory

As you can see, exactly which instruction the computer executes is pretty much up in the air—it is our job as a designer to figure out what we want it to do. For simplicity, let's design a computer that only implements the first two operations. Later on you can modify the design to accommodate the second two and then you'll be on your way to designing your own computer. But for now, let's stick with just the simpler instructions.

Question, though: what memory address will we load A from or store A to? Remember that we have to specify this address because our memory units require an address as input. While there are a great number of ways we can determine the address, for this first design we'll just assume that the address value we need is encoded within the 1's and 0's that make up the instruction itself (this is discussed further at the end of Sect. 20.2). We might write the shorthand for these instructions as follows:

- `LD A, N` Load the value from memory location `N` into register A
- `ST N, A` Store the value in register A into memory location `N`

(Note that as a general rule we usually write the *destination* of an operation on the left and the *sources* on the right when using shorthand.)

OK, so now we can bring information into the processor's register A and store information from the processor's register A to memory. But we have some other registers, R0-R7, also. How do we load values into those registers? We are going to need to be able to move information between A and the registers R0-R7. The following instructions take care of this need:

- Move the value stored in the register A to one of the registers R0 through R7
- Move the value from one of the registers R0 through R7 to A

In shorthand, these are written `MOV Rn, A` and `MOV A, Rn`, respectively (remember the *destination* of the move is on the left and the *source* of the move is on the right.)

To load a value from memory address `N` into R5 we have to use two instructions in sequence:

```
LD A, N
MOV R5, A
```

This may seem cumbersome but it keeps our computer design simpler because we don't have to implement more complicated instructions. With this example you can start to get a sense for the tradeoffs we have to make as designers: do we implement very complicated instructions which require a lot of hardware or do we implement simpler instructions which require less hardware? This is an introductory text, so we can't delve into these topics very deeply here. More advanced texts on computer architecture and design tackle these issues is more detail.

We'll stop here with our four data movement instructions. Let's move on now and actually do some calculations with the data we're now able to load and store.

Arithmetic and Logic Operations

So, now that we have the ability to load data into the processor let's actually *compute* something! We know how to build an Adder/Subtractor unit so let's have our computer do the following things:

- Add the value from A to the value in one of the registers R0-R7 and store the result back in A
- Subtract the value in one of R0-R7 from the value in A and store the result back in A

In shorthand, these would be `ADD A, Rn` and `SUB A, Rn`. Could we add/subtract values from two registers R0-R7 so that we'd have `ADD Rn, Rn`? Sure we could! But remember our tradeoffs: that would make for a more complicated design. Instead, we can use our two instructions to perform more involved calculations. Say we want to add R3 to R4 and store the result back in R7. While it may be great to have access to an instruction such as `ADD R7, R3, R4`, we can instead achieve this operation with the following sequence of simpler instructions:

```
MOV A, R3
ADD A, R4
MOV R7, A
```

Are there advantages to having one instruction be able to do this instead of having to use three? Yes, but there are advantages to simplicity as well. So, for now in this first design we will stick with the special register A as being a source and destination operand for all our arithmetic and logic operations. (In fact, A stands for *Accumulator*, as the register A *accumulates* the results of calculations.)

One more very useful thing is the ability to increment or decrement a register. Let's implement a decrement instruction. We can do this by making the special register A a counter and giving it a count down input called *dec*. We'll call this instruction `DEC A`.

OK, so now what's a *logic* operation? Remember Boolean algebra: instead of adding and subtracting we can XOR or AND or OR or NAND values together. Let's have our computer do the following things:

- calculate the XOR of A and a register R0-R7 and store the result back in A
- calculate the AND of A and a register R0-R7 and store the result back in A
- complement all the bits in A

We will call the first operation `XOR A, Rn`, the second `AND A, Rn` and the third `NOT A`.

If we build multipliers or dividers or all sorts of fancy things we can keep adding more types of arithmetic and logic operations to our computer. Let's start with these six arithmetic and logic instructions, however, and then you can add more later as you get the hang of the design process.

Program Control Instructions

Remember that our computer will execute instructions in the order it finds them in the memory. So the instruction at location 0 will be executed first, then the one at location 1, then the one at location 2, etc., until you get to the end of the memory and the Program Counter (PC) resets back to 0.

This is fine as far as it goes, but it turns out that we will need a bit more control over the sequence in which the instructions execute. If you have programming experience, you know that *loops* and *if* statements are required to do anything of consequence. If you don't have programming experience, then consider this a glimpse into your future: someday you will learn that *loops* and *if* statements are required to do anything of consequence.

So, we need to figure out how to have our datapath implement *loops* and *if* statements. Let's first consider what we want the *loop* to do:

- rather than executing the next instruction, execute the one I tell you to execute

OK, so we have to tell the computer to execute the instruction at the address we specify rather than the next address located in the Program Counter (PC). How do we do this? Again, as with our load/store instructions, there are a great many ways we could determine this address. For our first example we'll follow the convention

established earlier and assume the address of the instruction we want to execute next is located within the 1's and 0's which encode our instruction.

Now our shorthand for this instruction becomes

- `JUMP N` Execute instruction at address `N`

(Yes, it's common to refer to this operation as "jumping." While the computer, unfortunately, doesn't actually leap off the table we can think of it as the program "jumping" from one line to another within the program flow.)

Armed with our `JUMP` instruction we can now implement a *loop*, as the following sequence of instructions illustrates (line numbers are hex addresses where the instruction is stored in memory):

```
00    LD A, 34H
01    MOV R2, A
02    NOT A
03    ADD A, R4
04    MOV R7, A
05    JUMP 02H
```

This sequence of instructions will have our computer load the value from memory location 34H into A, move that value to R2, complement the bits in A, add them to the value in R4, store that value in R7, and then complement the bits in A again and so on. Forever. Never ends.

In order to end our loop we need an *if* statement. An *if* statement is more generally called a *conditional* statement. We need to come up with a *condition* that must be met in order for something to happen. In C++ or Java, we'd write things like *if(x > y)* or *if(x == 5)*. In principle, we can design our computer to use any condition we can think of. For this first simple design, let's use the condition $A = 0$. That is, we'll check whether the special register A is equal to 0. If A is zero, then we'll jump. If A is not zero, then we move on to the next instruction as normal.

We now have the instruction

- `JZ N` Jump to `N` only when the value in A is zero

Our loop from the previous example can now be written like this (line numbers are hex addresses):

```
00    LD A, 34H
01    MOV R2, A
02    NOT A
03    ADD A, R4
04    MOV R7, A
05    MOV A, R2
06    DEC A
07    MOV R2, A
08    JZ 0AH
09    JUMP 02H
0A    //continuation of the program
```

You can see now that the program will loop a number of times equal to the value stored in memory location 34H. Lines 05, 06, and 07 load a loop index from R2, decrement it, and then store it back in R2. When this loop index reaches 0 the `JZ` instruction will execute and bypass the `JUMP`. The location 0AH then contains the next instruction to be executed and we are freed from our infinite loop.

We can keep designing more and more sophisticated sorts of program control instructions. For now, however, let's just implement these two simple ones in our design.

Instruction Format

So we've decided on the operations we want our computer to implement. These operations are referred to as the *instruction set* of the computer. Our instruction set is as follows:

Data Movement	**Arithmetic/Logic**	**Control**
LD A, N	ADD A, Rn	JUMP N
ST N, A	SUB A, Rn	JZ N
MOV A, Rn	XOR A, Rn	
MOV Rn, A	AND A, Rn	
	NOT A	
	DEC A	

These instructions are written using letters which remind us of English words that describe what the instruction does. Since we need to store these instructions inside the memory unit of the computer system, and we can't store letters in computer memories, we need to find a way to convert these shorthand notations into 1's and 0's. The way we do this is by *encoding* the instruction in an *instruction format*.

Let's use the following instruction format:

Instruction	**Encoding**		
LD A, N	0000	0000	$N_7N_6N_5N_4N_3N_2N_1N_0$
ST N, A	0001	0000	$N_7N_6N_5N_4N_3N_2N_1N_0$
MOV A, Rn	0010	$0n_2n_1n_0$	00000000
MOV Rn, A	0011	$0n_2n_1n_0$	00000000
ADD A, Rn	1000	$0n_2n_1n_0$	00000000
SUB A, Rn	1100	$0n_2n_1n_0$	00000000
XOR A, Rn	1001	$0n_2n_1n_0$	00000000
AND A, Rn	1011	$0n_2n_1n_0$	00000000
NOT A	1110	0000	00000000
DEC A	1111	0000	00000000
JUMP N	0100	0000	$N_7N_6N_5N_4N_3N_2N_1N_0$
JZ N	0101	0000	$N_7N_6N_5N_4N_3N_2N_1N_0$

Wow, OK, so what does all this mean? First note that each instruction is encoded using 16 bits divided into three *fields*. The first field consists of the most significant four bits of the encoding and is called the *opcode*.

Notice that each instruction has a unique opcode. For example, the opcode for the `SUB` instruction is 1100 and the opcode for the `AND` instruction is 1011. The opcode field is important because it will tell the state machine controller which instruction is to be executed.

The second field consists of the next most significant four bits and is called the *register* field. This field denotes which register the instruction uses and is written as $n_2n_1n_0$. For example, if we want to encode the instruction `ADD A, R2` we would write `1000001000000000` because 010 is binary for 2. If we want to encode `MOV A, R7` we would write `0010011100000` because 111 is binary for 7. The instructions that don't use registers R0 through R7 have this field zeroed out.

The third field consists of the least significant eight bits of the instruction and is called the *memory* field. The load, store, and jump instructions require this field. It gives the address used in the operation. This field is zeroed out for the instructions which don't deal with memory addresses.

We will use these fields to help simplify and organize the design of our computer. They will come into play in both the datapath and controller designs.

If you look at the opcode field carefully you may notice some patterns. For example, you can see that the two most significant bits of the field determine which category the instruction falls into: 00 for data movement, 10 and 11 for arithmetic/logic, and 01 for control instructions. Assigning opcodes in this way will help us minimize the logic needed in the control unit. (You'll appreciate this more after working through Sect. 20.5.) There are other patterns there, too, for a discerning eye to pick up.

In principle we may assign any opcode we like to any instruction. We could number them in the order we think of them or number them alphabetically or roll dice to see what their opcodes should be. However, if we take some care with this process we can simplify our design later on. There is an art to this and as you gain more experience in computer design you'll see it in a more nuanced way. Right now, for your first design, it's okay if you come away with this just understanding that we have to encode instructions as 1's and 0's and that the table above gives us the 16-bit encodings we will use for each instruction. That's plenty to learn for your first design. Exactly how the encodings were determined, while fascinating to consider, is covered in other texts.

The Instruction Cycle

The only thing a computer does is *execute instructions sequentially*. That's it. Nothing more. To do this the computer needs to (1) keep track of what instruction is next in the program sequence and (2) issue the correct Control Word to the datapath to execute the appropriate instruction. This gives us an algorithm:

1. Load instruction from the memory address located in the PC into the IR
2. Execute the instruction located in the IR
3. Repeat from Step 1

Step 1 is called *Fetch* and step 2 is called *Execute* and we call this the *Instruction Cycle*.

Our task in designing a computer is to put together a datapath which will execute all our instructions and then to build a state machine controller which will implement the Instruction Cycle algorithm using our datapath.

That's it! That's computer design!

We've done each of these things throughout this text. In this chapter we'll put all of it together and construct an entire working computer from scratch.

Let's take a second to think about the Instruction Cycle in a bit more depth. Remember we implement algorithms by designing a state machine. Each state represents a step in the algorithm and the output of each state represents control signals which implement the desired operation on the datapath. The state machine is called the *controller* or the *control unit*.

Computer design is no different. Our Instruction Cycle is an algorithm and so we need a state machine controller to implement this algorithm with our datapath.

What will this state machine look like? Well, the first step of our algorithm is *Fetch* so we can begin there

The state diagramed in Fig. 20.1 is called *Fetch* and the output of the state is *CW_Fetch*, the Control Word for Fetch. You will need to figure out what the Control Word is that implements the Fetch step. This will depend on your datapath and your control signals. We've done this in the assignments, so this shouldn't be that hard at this point.

Now, what is the next state?

Well, isn't it *Execute?* Can't we just complete our state machine like seen in Fig. 20.2?

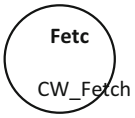


Fig. 20.1 The fetch stage

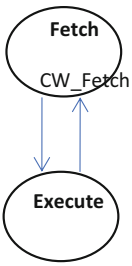


Fig. 20.2 Unsuccessful attempt at state diagram for Fetch-Execute controller

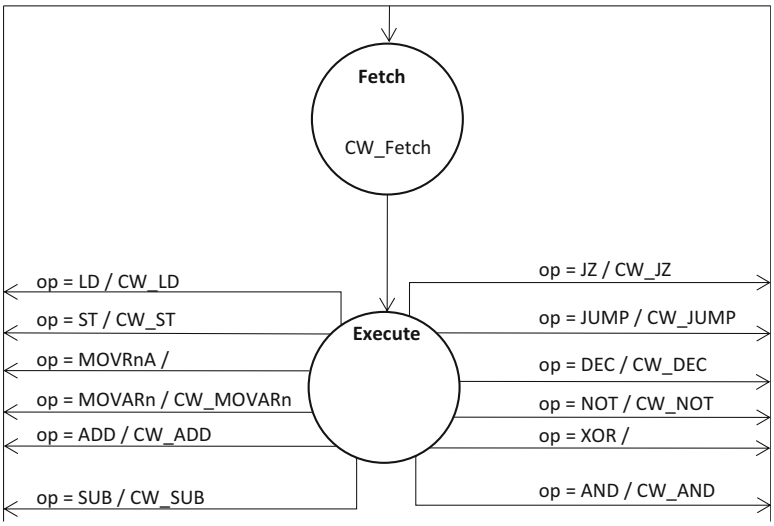


Fig. 20.3 Details of the execute stage

Unfortunately, it's not this simple. What will the output of Execute be? It can't be a single Control Word because the output will depend on which instruction is being executed, won't it? Our Fetch state has *Moore* type outputs: the output is the same regardless of input. The Execute state, however, requires *Mealy* type outputs—the output depends on what the input is. The opcode is the input and this opcode tells the controller which Control Word to send to the datapath. After all, the `ADD` instruction requires a different set of control signals than does the `MOV` instruction. So, really, our controller must look something like that given in Fig. 20.3.

Yeah, that's pretty cumbersome. We require a separate Control Word output for each possible opcode input. Think for a moment what the circuit for that design would look like. There would only be a single flip flop for the state memory and the output logic would be very complicated.

Actually, to be honest, for our very simple computer this is not so bad of a thing. We could implement this state machine and things would work out pretty well. The main problem with this simple setup is that it doesn't scale well to larger designs. There are a lot of reasons for this, all beyond the scope of this introductory text. For now, it suffices to understand that it's better to break our state machine controller down into smaller pieces so that each state handles more manageable chunks of the instruction set.

We would like to have six different Execute states, two for each category of instruction, as follows:

```
EXE1    LD, ST
EXE2    MOV
EXE3    ADD, SUB, XOR, AND
EXE4    NOT, DEC
EXE5    JUMP
EXE6    JZ
```

What we've done here is group the instructions based on what they require of the datapath. `LD` and `ST` are the only instructions which access memory. The two `MOV` instructions do their own thing. The arithmetic/logic instructions are divided into those which use one of the registers R0-R7 (`ADD`, `SUB`, `XOR`, `AND`) and those which use just the register A (`NOT`, `DEC`). The two control instructions are broken up because one is conditional and one is unconditional.

So, now our state machine control unit looks as shown in Fig. 20.4.

Perfect, right? We're done? Not so fast. While this feels like it should work there is a fatal error here.

What is it? Why won't this state machine work?

Look at the transitions out of Fetch. What are they based on? The opcode, right? Well, where is the opcode? It's part of the instruction which is stored in the Instruction Register (IR). Once you design the datapath you will need to supply a

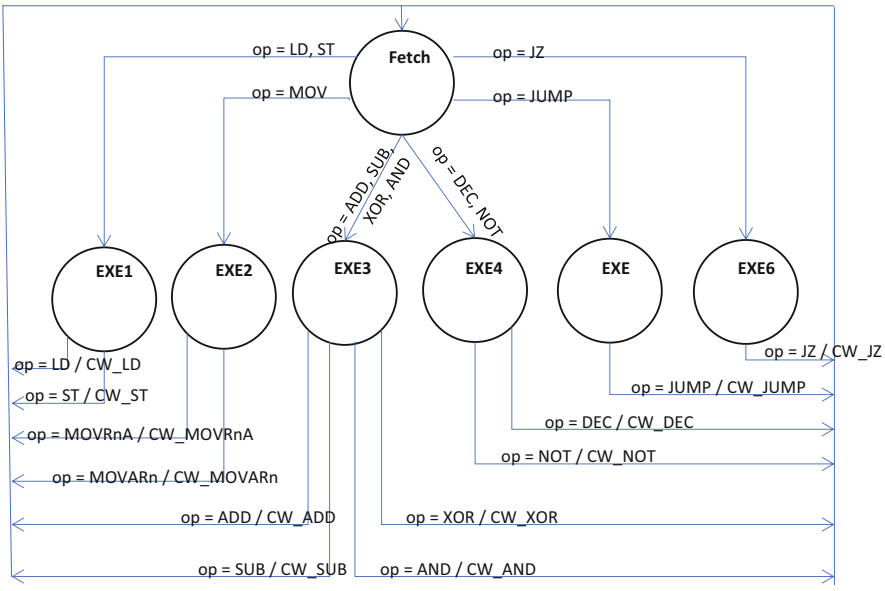


Fig. 20.4 Expanded execute stage

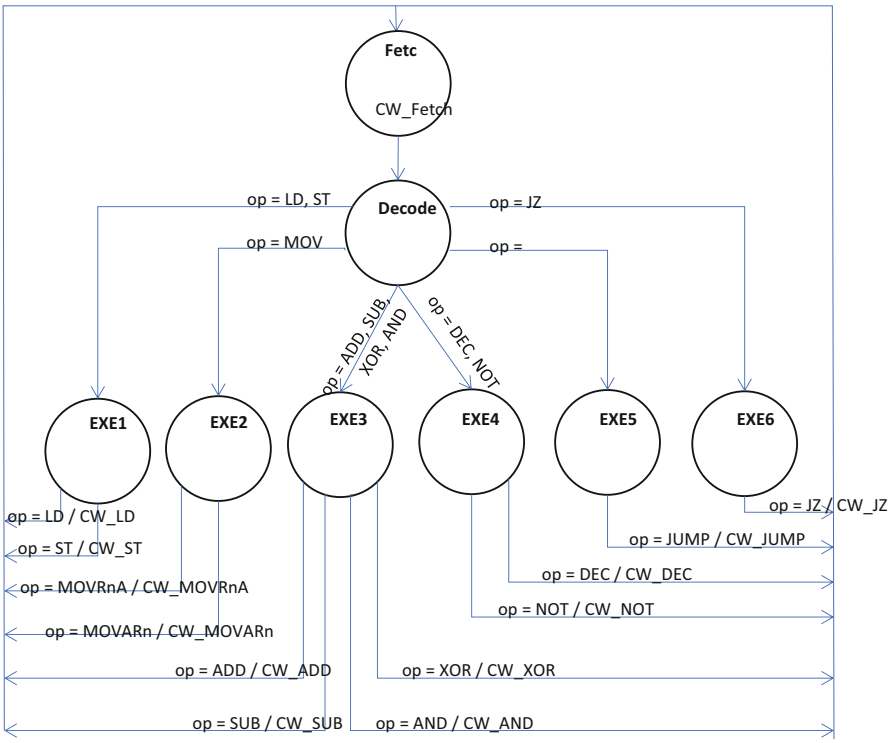


Fig. 20.5 Finalized state machine controller

status signal to the controller consisting of the opcode from the IR. So, why is it a problem? Why can't the Fetch state access the Instruction Register (IR)?

Think of hardware timing issues. . . . . *the registers update on the rising edge of the clock. The flip-flips tracking the states also transition on the rising edge of the clock. Therefore the opcode isn't even in the IR until the Fetch state is over!*

The value of the opcode can't be checked in Fetch! It must be checked in the next state.

This is key. Be sure to read through the above explanation again and again until you get this and understand why we need another state here. This is not obvious and of critical importance.

To fix this controller we need to introduce another state. In this state we will examine the opcode and *then* move to the appropriate Execute state.

We call this intermediate state *Decode*. Our final Instruction Cycle controller is shown in Fig. 20.5.

For our machine, all Decode does is give the controller a chance to look at the opcode and then move on to the appropriate Execute state. The output of Decode doesn't do anything—it just zeroes out all the loads and makes sure nothing gets overwritten. This may seem like a pretty weak state. After all, the Execute states

issue Control Words and implement awesome things like `XOR`s and `JZ`s. The truth of the matter is that the Decode state's power doesn't come into play in our very simple introductory computer design. In more complicated designs the Decode state is critical and will actually load registers, calculate values, and do all sorts of important and exciting things. Right now, however, we have a very simple instruction set which doesn't demand very much of Decode. That's fine as this is just an introductory example.

OK, so, we're pretty much done here now and are ready to design the datapath.

Datapath Design

Now that we have a list of things we want our computer to be able to do, we can design the datapath that will accommodate our requirements. Remember that this datapath needs to be able to implement the Instruction Cycle algorithm. That is, it needs to load instructions from memory (*Fetch*) and then *Decode* and *Execute* them.

For *Fetch*, remember that we store the address of the instruction in the Program Counter (PC) and we load the instruction into the Instruction Register (IR). So, start diagramming a datapath capable of implementing instruction *Fetch*: load the IR with the instruction located in the memory address stored in PC and then increment the PC so it is ready to get the next instruction when we get back to *Fetch*.

Decode requires the opcode be sent from the datapath as a status signal input to the controller.

For the *Execute* step of the Instruction Cycle, we need to be able to implement each of our twelve instructions. Remember that for status signals we need to output to the controller the opcode and a signal Z that tells us whether the A register is zero.

What follows are some comments which will help with your design. To get the most out of this exercise, I recommend you first work on the design yourself and then, once you think you have it, read the comments below. They will help you tighten your design and will be most useful to you if you've already worked on the datapath yourself.

Comment the First It's going to help us greatly if we use the fields of the instruction format within our datapath design. While the opcode needs to be sent to the controller as a status signal, the fields may also be used within the datapath itself to reduce the number of control signals needed. See if you can figure out how to do this and then review the next few comments for some, but not all, ways to accomplish such a reduction.

Comment the Second For your eight registers R0-R7 you should have a *dest* input to a 3:8 decoder which outputs the load signals for the registers, a *R_we* signal which enables the decoder, and a *src* signal for the 8:1 MUX which selects which of the registers we're going to use. We can eliminate both the *dest* and *src* signals by

noting that the register field of the instructions gives us this information. Run the $n_2n_1n_0$ from the IR to the decoder and the MUX. You should be left with the *R_we* signal and not have to worry about a *dest* and *src* control signal.

Comment the Third For the jump instructions, think about what it means to change the next instruction to be executed and make sure you make the appropriate connection on your datapath. This is very important for you to figure out.

Once the datapath is complete and you are sure that it can implement *fetch* as well as all twelve of our instructions, you are ready to design the controller. The fewer the number of signals you are using the easier this will be to do.

Also remember that although we draw the Mem unit as part of the datapath, it is really not actually inside the computer. We should really simply have memory address and data lines going out of the datapath and a memory data input line coming into the datapath, but it's easier for us to visualize initially if we include the Mem unit as its own box.

Control Unit Design

The datapath and controller work together to execute instructions sequentially. The datapath is able to execute all the instructions, but can only do so at the direction of the controller, whose states track the steps of the Instruction Cycle and whose outputs are the control signals necessary to determine what the datapath implements.

We've already discussed the design of the state machine controller. It's recommended that you review that material so that you fully understand the state diagram. Your task here is to finalize the missing details of that diagram by completing the state table.

Use the following state encoding:

Code	State
000	Fetch
001	Decode
010	EXE1
011	EXE2
100	EXE3
101	EXE4
110	EXE5
111	EXE6

The state table has been started for you. You will have to fill in the control signals appropriate for your datapath. Filled in already is the Next State information for the Fetch state and the first row of the Decode state. Fetch was easy: the next

state is always Decode regardless of what the input is. You will have to fill in the outputs to record the Control Word appropriate for the Fetch state for your datapath.

The Next State from Decode will be one of the EXE states. Which EXE state will depend on the opcode. You could list out all twelve opcodes one by one, but if you study the opcodes you will see some patterns that will make the table simpler by incorporating don't cares. You have an example of that if you write out the first row: notice that `LD` and `ST` transition to the same state EXE1 and they are the only instructions whose opcodes begin with 000.

The Decode state has a single Control Word that doesn't depend on the opcode. You'll need to fill it in for your datapath.

The remaining six states are all EXE states. The outputs for each of them will depend on the opcode. Again, the more don't cares you are able to put into the table the easier it will be to minimize the logic. The Next State for all of them is Fetch.

Finally, when the table is filled in, you will need to find the minimal Next State Logic. Use whatever method makes sense for you, but we are looking for minimal sums here.

The final step is to calculate the Output Logic. You may have a dozen or more control signals, so how about we just calculate the Output Logic for a few signals. Calculate the minimal equations for the logic for the LD_A, LD_IR, and R_we signals.

Now you're done! That's it! Congratulations, you've designed a computer!

Exercises

20.1 Draw the datapath that implements all the instructions detailed in this chapter as well as the extra Fetch and Decode steps required by the Instruction Cycle algorithm we discussed. Clearly label your status signals and your control signals. You may use any of the basic datapath elements as well as a memory unit.

20.2 The state diagram for the controller is found in Fig. 20.6. Fill in the state table and calculate the required Next State and Output logic.

20.3 Perform the required Next State and Output logic calculations from the state table you constructed in exercise 19.2. Clearly indicate your final logic equations.

20.4 Write a sequence of instructions that will perform the following calculation: R3 = R7 + R4 – R5 – 2.

20.5 Write a sequence of instructions which will multiply the value in R2 by the value in R4 and store the result in R5. Assume the values are unsigned and overflow is not a problem. (Hint: use a loop.)

20.6 How many more instructions could be added to the instruction set without changing the size of the *opcode* field? Explain your answer.

	Present State			Inputs					Next State			Outputs
				op_3	op_2	op_1	op_0	Z				
Fetch	0	0	0	x	x	x	x	x	0	0	1	
Decode	0	0	1	0	0	0	x	x	0	1	0	
EXE1	0	1	0									
EXE2	0	1	1									
EXE3	1	0	0									
EXE4	1	0	1									
EXE5	1	1	0									
EXE6	1	1	1									

Fig. 20.6 State table for exercise 19.2

20.7 In order to perform the calculation A = A + 5 we need to have a memory location with the constant 5 stored in it. This is annoying. Our instruction set can be improved if we allow the *memory* field to be used as a constant as well as an address. This gives us the following instruction:

`ADDi A, #N` Add the constant `N` to the value in A and store the result back in A

Explain how you would change your datapath to accommodate `ADDi`.

20.8 Similarly, the ability for the programmer to load values directly into registers R0-R7 is quite helpful, so we'd like to have the instruction

`MOVi Rn, #N` Store the constant `N` in the register `Rn`

Explain how to modify your datapath to accommodate `MOVi`.

20.9 Come up with another interesting instruction you think would be helpful, write its shorthand description, and explain how you'd change your datapath to accommodate it.

20.10 Write a sequence of instructions which will calculate the two's complement of the value located in R4 and store that value in R7. You may use any of our basic twelve instructions along with the `ADDi` and `MOVi` instructions from Questions 4 and 5 above.