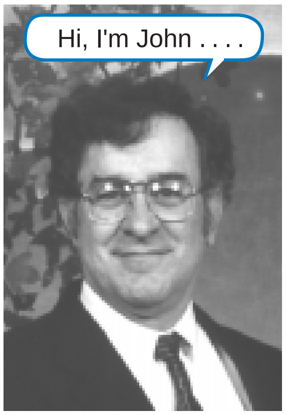


chapter 1

Introduction

Welcome to the world of digital design. Perhaps you're a computer science student who knows all about computer software and programming, but you're still trying to figure out how all that fancy hardware could possibly work. Or perhaps you're an electrical engineering student who already knows something about analog electronics and circuit design, but you wouldn't know a bit if it bit you. No matter. Starting from a fairly basic level, this book will show you how to design digital circuits and subsystems.

We'll give you the basic principles that you need to figure things out, and we'll give you lots of examples. Along with principles, we'll try to convey the flavor of real-world digital design by discussing current, practical considerations whenever possible. And I, the author, will often refer to myself as "we" in the hope that you'll be drawn in and feel that we're walking through the learning process together.

1.1 About Digital Design

Some people call it "logic design." That's OK, but ultimately the goal of design is to build systems. To that end, we'll cover a whole lot more in this text than just logic equations and theorems.

This book claims to be about principles and practices. Most of the principles that we present will continue to be important years from now; some may be applied in ways that have not even been discovered yet. As for practices, they may be a little different from what's presented here by the time you start working in the field, and they will certainly continue to change throughout your career. So you should treat the "practices" material in this book as a way to reinforce principles, and as a way to learn design methods by example.

One of the book's goals is to present enough about basic principles for you to know what's happening when you use software tools to turn the crank for you. The same basic principles can help you get to the root of problems when the tools happen to get in your way.

Listed in the box on this page, there are several key points that you should learn through your studies with this text. Most of these items probably make no sense to you right now, but you should come back and review them later.

Digital design is engineering, and engineering means "problem solving." My experience is that only 5%–10% of digital design is "the fun stuff"—the creative part of design, the flash of insight, the invention of a new approach. Much of the rest is just "turning the crank." To be sure, turning the crank is much easier now than it was 20 or even 10 years ago, but you still can't spend 100% or even 50% of your time on the fun stuff.

IMPORTANT THEMES IN DIGITAL DESIGN

- Good tools do not guarantee good design, but they help a lot by taking the pain out of doing things right.
- Digital circuits have analog characteristics.
- Know when to worry and when not to worry about the analog aspects of digital design.
- Always document your designs to make them understandable by yourself and others.
- Associate active levels with signal names and practice bubble-to-bubble logic design.
- Understand and use standard functional building blocks.
- Design for minimum cost at the system level, including your own engineering effort as part of the cost.
- State-machine design is like programming; approach it that way.
- Use programmable logic to simplify designs, reduce cost, and accommodate last-minute modifications.
- Avoid asynchronous design. Practice synchronous design until a better methodology comes along.
- Pinpoint the unavoidable asynchronous interfaces between different subsystems and the outside world, and provide reliable synchronizers.
- Catching a glitch in time saves nine.

Besides the fun stuff and turning the crank, there are many other areas in which a successful digital designer must be competent, including the following:

- *Debugging.* It's next to impossible to be a good designer without being a good troubleshooter. Successful debugging takes planning, a systematic approach, patience, and logic: if you can't discover where a problem *is*, find out where it *is not*!
- *Business requirements and practices.* A digital designer's work is affected by a lot of non-engineering factors, including documentation standards, component availability, feature definitions, target specifications, task scheduling, office politics, and going to lunch with vendors.
- *Risk-taking*. When you begin a design project you must carefully balance risks against potential rewards and consequences, in areas ranging from new-component selection (will it be available when I'm ready to build the first prototype?) to schedule commitments (will I still have a job if I'm late?).
- *Communication.* Eventually, you'll hand off your successful designs to other engineers, other departments, and customers. Without good communication skills, you'll never complete this step successfully. Keep in mind that communication includes not just transmitting but also receiving; learn to be a good listener!

In the rest of this chapter, and throughout the text, I'll continue to state some opinions about what's important and what is not. I think I'm entitled to do so as a moderately successful practitioner of digital design. Of course, you are always welcome to share your own opinions and experience (send email to `john@wakerly.com`).

1.2 Analog versus Digital

analog
digital

Analog devices and systems process time-varying signals that can take on any value across a continuous range of voltage, current, or other metric. So do *digital* circuits and systems; the difference is that we can pretend that they don't! A digital signal is modeled as taking on, at any time, only one of two discrete values, which we call *0* and *1* (or LOW and HIGH, FALSE and TRUE, negated and asserted, Sam and Fred, or whatever).

0
1

Digital computers have been around since the 1940s, and have been in widespread commercial use since the 1960s. Yet only in the past 10 to 20 years has the "digital revolution" spread to many other aspects of life. Examples of once-analog systems that have now "gone digital" include the following:

- Still pictures. The majority of cameras still use silver-halide film to record images. However, the increasing density of digital memory chips has allowed the development of digital cameras which record a picture as a

640×480 or larger array of pixels, where each pixel stores the intensities of its red, green and blue color components as 8 bits each. This large amount of data, over seven million bits in this example, may be processed and compressed into a format called JPEG with as little as 5% of the original storage size, depending on the amount of picture detail. So, digital cameras rely on both digital storage and digital processing.

- *Video recordings*. A digital versatile disc (DVD) stores video in a highly compressed digital format called MPEG-2. This standard encodes a small fraction of the individual video frames in a compressed format similar to JPEG, and encodes each other frame as the difference between it and the previous one. The capacity of a single-layer, single-sided DVD is about 35 billion bits, sufficient for about 2 hours of high-quality video, and a two-layer, double-sided disc has four times that capacity.
- *Audio recordings*. Once made exclusively by impressing analog waveforms onto vinyl or magnetic tape, audio recordings now commonly use digital compact discs (CDs). A CD stores music as a sequence of 16-bit numbers corresponding to samples of the original analog waveform, one sample per stereo channel every 22.7 microseconds. A full-length CD recording (73 minutes) contains over six billion bits of information.
- *Automobile carburetors*. Once controlled strictly by mechanical linkages (including clever "analog" mechanical devices that sensed temperature, pressure, etc.), automobile engines are now controlled by embedded microprocessors. Various electronic and electromechanical sensors convert engine conditions into numbers that the microprocessor can examine to determine how to control the flow of fuel and oxygen to the engine. The microprocessor's output is a time-varying sequence of numbers that operate electromechanical actuators which, in turn, control the engine.
- *The telephone system*. It started out a hundred years ago with analog microphones and receivers connected to the ends of a pair of copper wires (or was it string?). Even today, most homes still use analog telephones, which transmit analog signals to the phone company's central office (CO). However, in the majority of COs, these analog signals are converted into a digital format before they are routed to their destinations, be they in the same CO or across the world. For many years the private branch exchanges (PBXs) used by businesses have carried the digital format all the way to the desktop. Now many businesses, COs, and traditional telephony service providers are converting to integrated systems that combine digital voice with data traffic over a single IP (Internet Protocol) network.
- *Traffic lights*. Stop lights used to be controlled by electromechanical timers that would give the green light to each direction for a predetermined amount of time. Later, relays were used in controllers that could activate

the lights according to the pattern of traffic detected by sensors embedded in the pavement. Today's controllers use microprocessors, and can control the lights in ways that maximize vehicle throughput or, in some California cities, frustrate drivers in all kinds of creative ways.

- *Movie effects.* Special effects used to be made exclusively with miniature clay models, stop action, trick photography, and numerous overlays of film on a frame-by-frame basis. Today, spaceships, bugs, other-worldly scenes, and even babies from hell (in Pixar's animated feature *Tin Toy*) are synthesized entirely using digital computers. Might the stunt man or woman someday no longer be needed, either?

The electronics revolution has been going on for quite some time now, and the "solid-state" revolution began with analog devices and applications like transistors and transistor radios. So why has there now been a *digital* revolution? There are in fact many reasons to favor digital circuits over analog ones:

- *Reproducibility of results.* Given the same set of inputs (in both value and time sequence), a properly designed digital circuit always produces exactly the same results. The outputs of an analog circuit vary with temperature, power-supply voltage, component aging, and other factors.
- *Ease of design.* Digital design, often called "logic design," is logical. No special math skills are needed, and the behavior of small logic circuits can be visualized mentally without any special insights about the operation of capacitors, transistors, or other devices that require calculus to model.
- *Flexibility and functionality.* Once a problem has been reduced to digital form, it can be solved using a set of logical steps in space and time. For example, you can design a digital circuit that scrambles your recorded voice so that it is absolutely indecipherable by anyone who does not have your "key" (password), but can be heard virtually undistorted by anyone who does. Try doing that with an analog circuit.
- *Programmability.* You're probably already quite familiar with digital computers and the ease with which you can design, write, and debug programs for them. Well, guess what? Much of digital design is carried out today by writing programs, too, in *hardware description languages (HDLs)*. These languages allow both structure and function of a digital circuit to be specified or *modeled*. Besides a compiler, a typical HDL also comes with simulation and synthesis programs. These software tools are used to test the hardware model's behavior before any real hardware is built, and then synthesize the model into a circuit in a particular component technology.

hardware description language (HDL)

hardware model

- *Speed.* Today's digital devices are very fast. Individual transistors in the fastest integrated circuits can switch in less than 10 picoseconds, and a complete, complex device built from these transistors can examine its

SHORT TIMES A *microsecond (μsec)* is 10^{-6} second. A *nanosecond (ns)* is just 10^{-9} second, and a *picosecond (ps)* is 10^{-12} second. In a vacuum, light travels about a foot in a nanosecond, and an inch in 85 picoseconds. With individual transistors in the fastest integrated circuits now switching in less than 10 picoseconds, the speed-of-light delay between these transistors across a half-inch-square silicon chip has become a limiting factor in circuit design.

inputs and produce an output in less than 2 nanoseconds. This means that such a device can produce 500 million or more results per second.

- *Economy.* Digital circuits can provide a lot of functionality in a small space. Circuits that are used repetitively can be "integrated" into a single "chip" and mass-produced at very low cost, making possible throw-away items like calculators, digital watches, and singing birthday cards. (You may ask, "Is this such a good thing?" Never mind!)
- *Steadily advancing technology.* When you design a digital system, you almost always know that there will be a faster, cheaper, or otherwise better technology for it in a few years. Clever designers can accommodate these expected advances during the initial design of a system, to forestall system obsolescence and to add value for customers. For example, desktop computers often have "expansion sockets" to accommodate faster processors or larger memories than are available at the time of the computer's introduction.

So, that's enough of a sales pitch on digital design. The rest of this chapter will give you a bit more technical background to prepare you for the rest of the book.

1.3 Digital Devices

gate

The most basic digital devices are called *gates* and no, they were not named after the founder of a large software company. Gates originally got their name from their function of allowing or retarding ("gating") the flow of digital information. In general, a gate has one or more inputs and produces an output that is a function of the current input value(s). While the inputs and outputs may be analog conditions such as voltage, current, even hydraulic pressure, they are modeled as taking on just two discrete values, 0 and 1.

AND gate

Figure 1-1 shows symbols for the three most important kinds of gates. A 2-input *AND gate*, shown in (a), produces a 1 output if both of its inputs are 1; otherwise it produces a 0 output. The figure shows the same gate four times, with the four possible combinations of inputs that may be applied to it and the result-

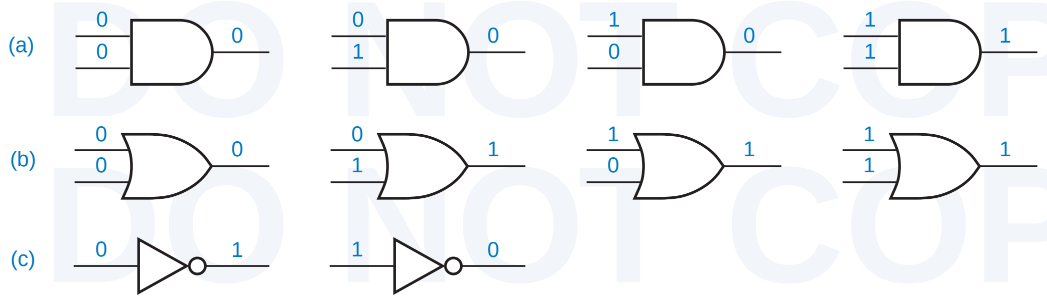


Figure 1-1 Digital devices: (a) AND gate; (b) OR gate; (c) NOT gate or inverter.

ing outputs. A gate is called a *combinational* circuit because its output depends only on the current input combination.

combinational

A 2-input *OR gate*, shown in (b), produces a 1 output if one or both of its inputs are 1; it produces a 0 output only if both inputs are 0. Once again, there are four possible input combinations, resulting in the outputs shown in the figure.

OR gate

A *NOT gate*, more commonly called an *inverter*, produces an output value that is the opposite of the input value, as shown in (c).

NOT gate
inverter

We called these three gates the most important for good reason. Any digital function can be realized using just these three kinds of gates. In Chapter 3 we'll show how gates are realized using transistor circuits. You should know, however, that gates have been built or proposed using other technologies, such as relays, vacuum tubes, hydraulics, and molecular structures.

A *flip-flop* is a device that stores either a 0 or 1. The *state* of a flip-flop is the value that it currently stores. The stored value can be changed only at certain times determined by a "clock" input, and the new value may further depend on the flip-flop's current state and its "control" inputs. A flip-flop can be built from a collection of gates hooked up in a clever way, as we'll show in Section 7.2.

flip-flop
state

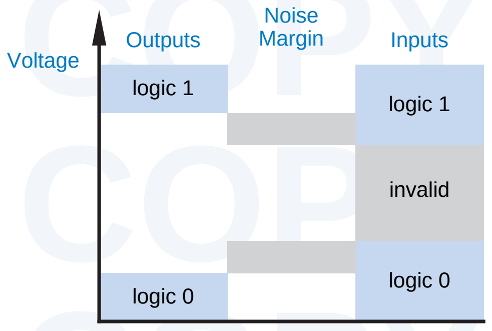
A digital circuit that contains flip-flops is called a *sequential circuit* because its output at any time depends not only on its current input, but also on the past sequence of inputs that have been applied to it. In other words, a sequential circuit has *memory* of past events.

sequential circuit
memory

1.4 Electronic Aspects of Digital Design

Digital circuits are not exactly a binary version of alphabet soup—with all due respect to Figure 1-1, they don't have little 0s and 1s floating around in them. As we'll see in Chapter 3, digital circuits deal with analog voltages and currents, and are built with analog components. The "digital abstraction" allows analog behavior to be ignored in most cases, so circuits can be modeled as if they really did process 0s and 1s.

Figure 1-2
Logic values and noise margins.



One important aspect of the digital abstraction is to associate a *range* of analog values with each logic value (0 or 1). As shown in Figure 1-2, a typical gate is not guaranteed to have a precise voltage level for a logic 0 output. Rather, it may produce a voltage somewhere in a range that is a *subset* of the range guaranteed to be recognized as a 0 by other gate inputs. The difference between the range boundaries is called *noise margin*—in a real circuit, a gate's output can be corrupted by this much noise and still be correctly interpreted at the inputs of other gates.

noise margin

Behavior for logic 1 outputs is similar. Note in the figure that there is an "invalid" region between the input ranges for logic 0 and logic 1. Although any given digital device operating at a particular voltage and temperature will have a fairly well defined boundary (or threshold) between the two ranges, different devices may have different boundaries. Still, all properly operating devices have their boundary *somewhere* in the "invalid" range. Therefore, any signal that is within the defined ranges for 0 and 1 will be interpreted identically by different devices. This characteristic is essential for reproducibility of results.

It is the job of an *electronic* circuit designer to ensure that logic gates produce and recognize logic signals that are within the appropriate ranges. This is an analog circuit-design problem; we touch upon some aspects of this in Chapter 3. It is not possible to design a circuit that has the desired behavior under every possible condition of power-supply voltage, temperature, loading, and other factors. Instead, the electronic circuit designer or device manufacturer provides *specifications* that define the conditions under which correct behavior is guaranteed.

specifications

As a *digital* designer, then, you need not delve into the detailed analog behavior of a digital device to ensure its correct operation. Rather, you need only examine enough about the device's operating environment to determine that it is operating within its published specifications. Granted, some analog knowledge is needed to perform this examination, but not nearly what you'd need to design a digital device starting from scratch. In Chapter 3, we'll give you just what you need.

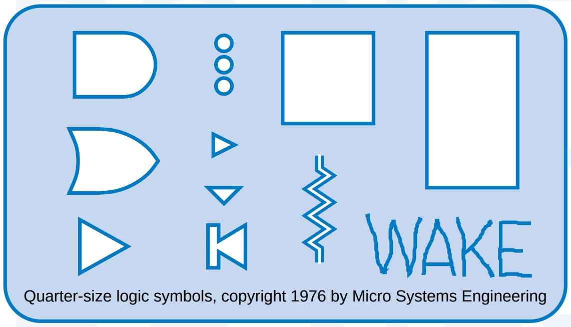

WAKE
Quarter-size logic symbols, copyright 1976 by Micro Systems Engineering

Figure 1-3
A logic-design template.

1.5 Software Aspects of Digital Design

Digital design need not involve any software tools. For example, Figure 1-3 shows the primary tool of the "old school" of digital design—a plastic template for drawing logic symbols in schematic diagrams by hand (the designer's name was engraved into the plastic with a soldering iron).

Today, however, software tools are an essential part of digital design. Indeed, the availability and practicality of hardware description languages (HDLs) and accompanying circuit simulation and synthesis tools have changed the entire landscape of digital design over the past several years. We'll make extensive use of HDLs throughout this book.

computer-aided design (CAD)

In *computer-aided design (CAD)* various software tools improve the designer's productivity and help to improve the correctness and quality of designs. In a competitive world, the use of software tools is mandatory to obtain high-quality results on aggressive schedules. Important examples of software tools for digital design are listed below:

- *Schematic entry.* This is the digital designer's equivalent of a word processor. It allows schematic diagrams to be drawn "on-line," instead of with paper and pencil. The more advanced schematic-entry programs also check for common, easy-to-spot errors, such as shorted outputs, signals that don't go anywhere, and so on. Such programs are discussed in greater detail in Section 12.1.
- *HDLs.* Hardware description languages, originally developed for circuit modeling, are now being used more and more for hardware *design*. They can be used to design anything from individual function modules to large, multi-chip digital systems. We'll introduce two HDLs, ABEL and VHDL, at the end of Chapter 4, and we'll provide examples in both languages in the chapters that follow.
- *HDL compilers, simulators, and synthesis tools.* A typical HDL software package contains several components. In a typical environment, the designer writes a text-based "program," and the HDL compiler analyzes

the program for syntax errors. If it compiles correctly, the designer has the option of handing it over to a synthesis tool that creates a corresponding circuit design targeted to a particular hardware technology. Most often, before synthesis the designer will use the compiler's results as input to a "simulator" to verify the behavior of the design.

- *Simulators*. The design cycle for a customized, single-chip digital integrated circuit is long and expensive. Once the first chip is built, it's very difficult, often impossible, to debug it by probing internal connections (they are really tiny), or to change the gates and interconnections. Usually, changes must be made in the original design database and a new chip must be manufactured to incorporate the required changes. Since this process can take months to complete, chip designers are highly motivated to "get it right" (or almost right) on the first try. Simulators help designers predict the electrical and functional behavior of a chip without actually building it, allowing most if not all bugs to be found before the chip is fabricated.
- Simulators are also used in the design of "programmable logic devices," introduced later, and in the overall design of systems that incorporate many individual components. They are somewhat less critical in this case because it's easier for the designer to make changes in components and interconnections on a printed-circuit board. However, even a little bit of simulation can save time by catching simple but stupid mistakes.
- *Test benches*. Digital designers have learned how to formalize circuit simulation and testing into software environments called "test benches." The idea is to build a set of programs around a design to automatically exercise its functions and check both its functional and its timing behavior. This is especially useful when small design changes are made—the test bench can be run to ensure that bug fixes or "improvements" in one area do not break something else. Test-bench programs may be written in the same HDL as the design itself, in C or C++, or in combination of languages including scripting languages like PERL.
- *Timing analyzers and verifiers*. The time dimension is very important in digital design. All digital circuits take time to produce a new output value in response to an input change, and much of a designer's effort is spent ensuring that such output changes occur quickly enough (or, in some cases, not too quickly). Specialized programs can automate the tedious task of drawing timing diagrams and specifying and verifying the timing relationships between different signals in a complex system.
- *Word processors*. Let's not forget the lowly text editor and word processor. These tools are obviously useful for creating the source code for HDL-based designs, but they have an important use in every design—to create documentation!

PROGRAMMABLE LOGIC DEVICES VERSUS SIMULATION

Later in this book you'll learn how programmable logic devices (PLDs) and field-programmable gate arrays (FPGAs) allow you to design a circuit or subsystem by writing a sort of program. PLDs and FPGAs are now available with up to millions of gates, and the capabilities of these technologies are ever increasing. If a PLD- or FPGA-based design doesn't work the first time, you can often fix it by changing the program and physically reprogramming the device, without changing any components or interconnections at the system level. The ease of prototyping and modifying PLD- and FPGA-based systems can eliminate the need for simulation in board-level design; simulation is required only for chip-level designs.

The most widely held view in industry trends says that as chip technology advances, more and more design will be done at the chip level, rather than the board level. Therefore, the ability to perform complete and accurate simulation will become increasingly important to the typical digital designer.

However, another view is possible. If we extrapolate trends in PLD and FPGA capabilities, in the next decade we will witness the emergence of devices that include not only gates and flip-flops as building blocks, but also higher-level functions such as processors, memories, and input/output controllers. At this point, most digital designers will use complex on-chip components and interconnections whose basic functions have already been tested by the device manufacturer.

In this future view, it is still possible to misapply high-level programmable functions, but it is also possible to fix mistakes simply by changing a program; detailed simulation of a design before simply "trying it out" could be a waste of time. Another, compatible view is that the PLD or FPGA is merely a full-speed simulator for the program, and this full-speed simulator is what gets shipped in the product!

Does this extreme view have any validity? To guess the answer, ask yourself the following question. How many software programmers do you know who debug a new program by "simulating" its operation rather than just trying it out?

In any case, modern digital systems are much too complex for a designer to have any chance of testing every possible input condition, with or without simulation. As in software, correct operation of digital systems is best accomplished through practices that ensure that the systems are "correct by design." It is a goal of this text to encourage such practices.

In addition to using the tools above, designers may sometimes write specialized programs in high-level languages like C or C++, or scripts in languages like PERL, to solve particular design problems. For example, Section 11.1 gives a few examples of C programs that generate the "truth tables" for complex combinational logic functions.

Although CAD tools are important, they don't make or break a digital designer. To take an analogy from another field, you couldn't consider yourself to be a great writer just because you're a fast typist or very handy with a word processor. During your study of digital design, be sure to learn and use all the tools that are available to you, such as schematic-entry programs, simulators, and HDL compilers. But remember that learning to use tools is no guarantee that you'll be able to produce good results. Please pay attention to what you're producing with them!

1.6 Integrated Circuits

integrated circuit (IC)

A collection of one or more gates fabricated on a single silicon chip is called an *integrated circuit (IC)*. Large ICs with tens of millions of transistors may be half an inch or more on a side, while small ICs may be less than one-tenth of an inch on a side.

wafer

die

Regardless of its size, an IC is initially part of a much larger, circular *wafer*, up to ten inches in diameter, containing dozens to hundreds of replicas of the same IC. All of the IC chips on the wafer are fabricated at the same time, like pizzas that are eventually sold by the slice, except in this case, each piece (IC chip) is called a *die*. After the wafer is fabricated, the dice are tested in place on the wafer and defective ones are marked. Then the wafer is sliced up to produce the individual dice, and the marked ones are discarded. (Compare with the pizza-maker who sells all the pieces, even the ones without enough pepperoni!) Each unmarked die is mounted in a package, its pads are connected to the package pins, and the packaged IC is subjected to a final test and is shipped to a customer.

IC

Some people use the term "IC" to refer to a silicon die. Some use "chip" to refer to the same thing. Still others use "IC" or "chip" to refer to the combination of a silicon die and its package. Digital designers tend to use the two terms interchangeably, and they really don't care what they're talking about. They don't require a precise definition, since they're only looking at the functional and electrical behavior of these things. In the balance of this text, we'll use the term *IC* to refer to a packaged die.

A DICEY DECISION

A reader of the second edition wrote to me to collect a $5 reward for pointing out my "glaring" misuse of "dice" as the plural of "die." According to the dictionary, she said, the plural form of "die" is "dice" *only* when describing those little cubes with dots on each side; otherwise it's "dies," and she produced the references to prove it.

Being stubborn, I asked my friends at the *Microprocessor Report* about this issue. According to the editor,

> There is, indeed, much dispute over this term. We actually stopped using the term "dice" in *Microprocessor Report* more than four years ago. I actually prefer the plural "die," … but perhaps it is best to avoid using the plural whenever possible.

So there you have it, even the experts don't agree with the dictionary! Rather than cop out, I boldly chose to use "dice" anyway, by rolling the dice.

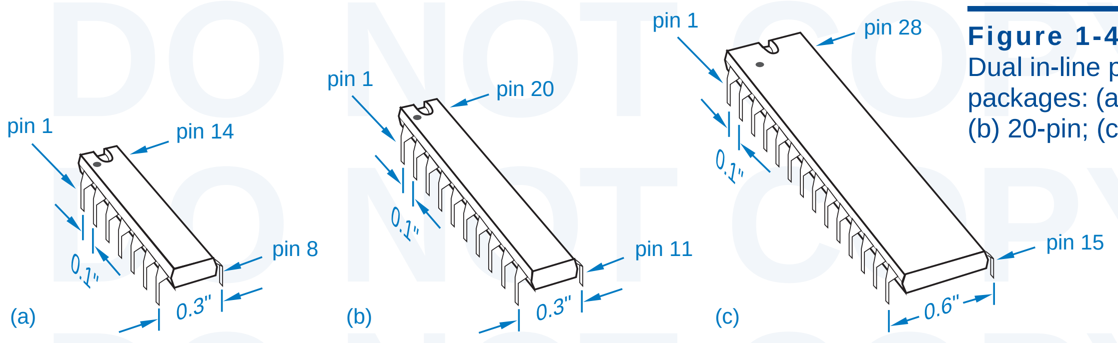


Figure 1-4 Dual in-line pin (DIP) packages: (a) 14-pin; (b) 20-pin; (c) 28-pin.

In the early days of integrated circuits, ICs were classified by size—small, medium, or large—according to how many gates they contained. The simplest type of commercially available ICs are still called *small-scale integration (SSI)*, and contain the equivalent of 1 to 20 gates. SSI ICs typically contain a handful of gates or flip-flops, the basic building blocks of digital design.

small-scale integration (SSI)

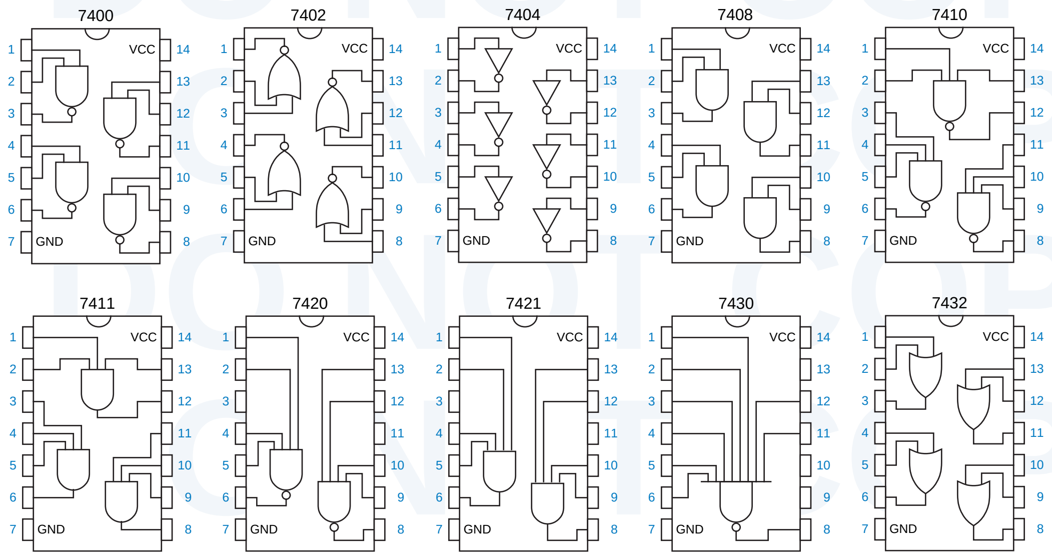
The SSI ICs that you're likely to encounter in an educational lab come in a 14-pin *dual in-line-pin (DIP)* package. As shown in Figure 1-4(a), the spacing between pins in a column is 0.1 inch and the spacing between columns is 0.3 inch. Larger DIP packages accommodate functions with more pins, as shown in (b) and (c). A *pin diagram* shows the assignment of device signals to package pins, or *pinout*. Figure 1-5 shows the pin diagrams for a few common SSI ICs. Such diagrams are used only for mechanical reference, when a designer needs to determine the pin numbers for a particular IC. In the schematic diagram for a

dual in-line-pin (DIP) package

pin diagram

pinout

Figure 1-5 Pin diagrams for a few 7400-series SSI ICs.



TINY-SCALE INTEGRATION

In the coming years, perhaps the most popular remaining use of SSI and MSI, especially in DIP packages, will be in educational labs. These devices will afford students the opportunity to "get their hands" dirty by "breadboarding" and wiring up simple circuits in the same way that their professors did years ago.

However, much to my surprise and delight, a segment of the IC industry has actually gone *down*scale from SSI in the past few years. The idea has been to sell individual logic gates in very small packages. These devices handle simple functions that are sometimes needed to match larger-scale components to a particular design, or in some cases they are used to work around bugs in the larger-scale components or their interfaces.

An example of such an IC is Motorola's 74VHC1G00. This chip is a single 2-input NAND gate housed in a 5-pin package (power, ground, two inputs, and one output). The entire package, including pins, measures only 0.08 inches on a side, and is only 0.04 inches high! Now that's what I would call "tiny-scale integration"!

digital circuit, pin diagrams are not used. Instead, the various gates are grouped functionally, as we'll show in Section 5.1.

Although SSI ICs are still sometimes used as "glue" to tie together larger-scale elements in complex systems, they have been largely supplanted by programmable logic devices, which we'll study in Sections 5.3 and 8.3.

medium-scale integration (MSI)

The next larger commercially available ICs are called *medium-scale integration (MSI)*, and contain the equivalent of about 20 to 200 gates. An MSI IC typically contains a functional building block, such as a decoder, register, or counter. In Chapters 5 and 8, we'll place a strong emphasis on these building blocks. Even though the use of discrete MSI ICs is declining, the equivalent building blocks are used extensively in the design of larger ICs.

large-scale integration (LSI)

Large-scale integration (LSI) ICs are bigger still, containing the equivalent of 200 to 200,000 gates or more. LSI parts include small memories, microprocessors, programmable logic devices, and customized devices.

STANDARD LOGIC FUNCTIONS

Many standard "high-level" functions appear over and over as building blocks in digital design. Historically, these functions were first integrated in MSI circuits. Subsequently, they have appeared as components in the "macro" libraries for ASIC design, as "standard cells" in VLSI design, as "canned" functions in PLD programming languages, and as library functions in hardware-description languages such as VHDL.

Standard logic functions are introduced in Chapters 5 and 8 as 74-series MSI parts, as well as in HDL form. The discussion and examples in these chapters provide a basis for understanding and using these functions in any form.

The dividing line between LSI and *very large-scale integration (VLSI)* is fuzzy, and tends to be stated in terms of transistor count rather than gate count. Any IC with over 1,000,000 transistors is definitely VLSI, and that includes most microprocessors and memories nowadays, as well as larger programmable logic devices and customized devices. In 1999, the VLSI ICs as large as 50 million transistors were being designed.

very large-scale integration (VLSI)

1.7 Programmable Logic Devices

There are a wide variety of ICs that can have their logic function "programmed" into them after they are manufactured. Most of these devices use technology that also allows the function to be *re*programmed, which means that if you find a bug in your design, you may be able to fix it without physically replacing or rewiring the device. In this book, we'll frequently refer to the design opportunities and methods for such devices.

Historically, *programmable logic arrays (PLAs)* were the first programmable logic devices. PLAs contained a two-level structure of AND and OR gates with user-programmable connections. Using this structure, a designer could accommodate any logic function up to a certain level of complexity using the well-known theory of logic synthesis and minimization that we'll present in Chapter 4.

programmable logic array (PLA)

PLA structure was enhanced and PLA costs were reduced with the introduction of *programmable array logic (PAL) devices*. Today, such devices are generically called programmable logic devices (PLDs), and are the "MSI" of the programmable logic industry. We'll have a lot to say about PLD architecture and technology in Sections 5.3 and 8.3.

programmable array logic (PAL) device

programmable logic device (PLD)

The ever-increasing capacity of integrated circuits created an opportunity for IC manufacturers to design larger PLDs for larger digital-design applications. However, for technical reasons that we'll discuss in \secref{CPLDs}, the basic two-level AND-OR structure of PLDs could not be scaled to larger sizes. Instead, IC manufacturers devised *complex PLD (CPLD)* architectures to achieve the required scale. A typical CPLD is merely a collection of multiple PLDs and an interconnection structure, all on the same chip. In addition to the individual PLDs, the on-chip interconnection structure is also programmable, providing a rich variety of design possibilities. CPLDs can be scaled to larger sizes by increasing the number of individual PLDs and the richness of the interconnection structure on the CPLD chip.

complex PLD (CPLD)

At about the same time that CPLDs were being invented, other IC manufacturers took a different approach to scaling the size of programmable logic chips. Compared to a CPLD, a field-programmable gate arrays (FPGA) contains a much larger number of smaller individual logic blocks, and provides a large, distributed interconnection structure that dominates the entire chip. Figure 1-6 illustrates the difference between the two chip-design approaches.

field-programmable gate array (FPGA)

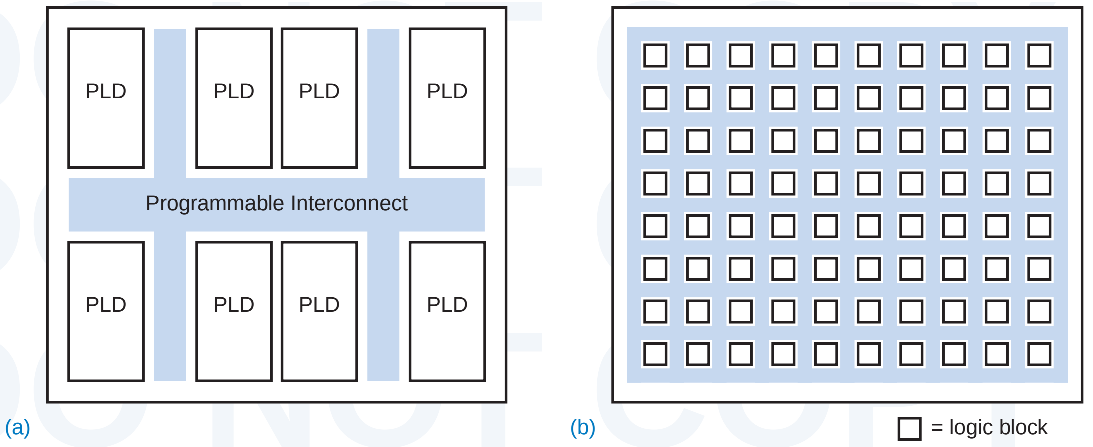


Figure 1-6 Large programmable-logic-device scaling approaches: (a) CPLD; (b) FPGA.

Proponents of one approach or the other used to get into “religious” arguments over which way was better, but the largest manufacturer of large programmable logic devices, Xilinx Corporation, acknowledges that there is a place for both approaches and manufactures both types of devices. What’s more important than chip architecture is that both approaches support a style of design in which products can be moved from design concept to prototype and production in a very period of time short time.

Also important in achieving short “time-to-market” for all kinds of PLD-based products is the use of HDLs in their design. Languages like ABEL and VHDL, and their accompanying software tools, allow a design to be compiled, synthesized, and downloaded into a PLD, CPLD, or FPGA literally in minutes. The power of highly structured, hierarchical languages like VHDL is especially important in helping designers utilize the hundreds of thousands or millions of gates that are provided in the largest CPLDs and FPGAs.

1.8 Application-Specific ICs

Perhaps the most interesting developments in IC technology for the average digital designer are not the ever-increasing chip sizes, but the ever-increasing opportunities to “design your own chip.” Chips designed for a particular, limited product or application are called *semicustom ICs* or *application-specific ICs (ASICs)*. ASICs generally reduce the total component and manufacturing cost of a product by reducing chip count, physical size, and power consumption, and they often provide higher performance.

semicustom IC

application-specific IC (ASIC)

The *nonrecurring engineering (NRE) cost* for designing an ASIC can exceed the cost of a discrete design by $5,000 to $250,000 or more. NRE charges are paid to the IC manufacturer and others who are responsible for designing the

nonrecurring engineering (NRE) cost

internal structure of the chip, creating tooling such as the metal masks for manufacturing the chips, developing tests for the manufactured chips, and actually making the first few sample chips.

The NRE cost for a typical, medium-complexity ASIC with about 100,000 gates is $30–$50,000. An ASIC design normally makes sense only when the NRE cost can be offset by the per-unit savings over the expected sales volume of the product.

custom LSI

The NRE cost to design a *custom LSI* chip—a chip whose functions, internal architecture, and detailed transistor-level design is tailored for a specific customer—is very high, $250,000 or more. Thus, full custom LSI design is done only for chips that have general commercial application or that will enjoy very high sales volume in a specific application (e.g., a digital watch chip, a network interface, or a bus-interface circuit for a PC).

standard cells

standard-cell design

To reduce NRE charges, IC manufacturers have developed libraries of *standard cells* including commonly used MSI functions such as decoders, registers, and counters, and commonly used LSI functions such as memories, programmable logic arrays, and microprocessors. In a *standard-cell design*, the logic designer interconnects functions in much the same way as in a multichip MSI/LSI design. Custom cells are created (at added cost, of course) only if absolutely necessary. All of the cells are then laid out on the chip, optimizing the layout to reduce propagation delays and minimize the size of the chip. Minimizing the chip size reduces the per-unit cost of the chip, since it increases the number of chips that can be fabricated on a single wafer. The NRE cost for a standard-cell design is typically on the order of $150,000.

gate array

Well, $150,000 is still a lot of money for most folks, so IC manufacturers have gone one step further to bring ASIC design capability to the masses. A *gate array* is an IC whose internal structure is an array of gates whose interconnections are initially unspecified. The logic designer specifies the gate types and interconnections. Even though the chip design is ultimately specified at this very low level, the designer typically works with “macrocells,” the same high-level functions used in multichip MSI/LSI and standard-cell designs; software expands the high-level design into a low-level one.

The main difference between standard-cell and gate-array design is that the macrocells and the chip layout of a gate array are not as highly optimized as those in a standard-cell design, so the chip may be 25% or more larger, and therefore may cost more. Also, there is no opportunity to create custom cells in the gate-array approach. On the other hand, a gate-array design can be completed faster and at lower NRE cost, ranging from about $5000 (what you’re told initially) to $75,000 (what you find you’ve spent when you’re all done).

The basic digital design methods that you’ll study throughout this book apply very well to the functional design of ASICs. However, there are additional opportunities, constraints, and steps in ASIC design, which usually depend on the particular ASIC vendor and design environment.

1.9 Printed-Circuit Boards

printed-circuit board (PCB)

printed-wiring board (PWB)

An IC is normally mounted on a *printed-circuit board (PCB)* [or *printed-wiring board (PWB)*] that connects it to other ICs in a system. The multilayer PCBs used in typical digital systems have copper wiring etched on multiple, thin layers of fiberglass that are laminated into a single board about 1/16 inch thick.

PCB traces

mil

fine-line

Individual wire connections, or *PCB traces* are usually quite narrow, 10 to 25 mils in typical PCBs. (A *mil* is one-thousandth of an inch.) In *fine-line* PCB technology, the traces are extremely narrow, as little as 4 mils wide with 4-mil spacing between adjacent traces. Thus, up to 125 connections may be routed in a one-inch-wide band on a single layer of the PCB. If higher connection density is needed, then more layers are used.

surface-mount technology (SMT)

Most of the components in modern PCBs use *surface-mount technology (SMT)*. Instead of having the long pins of DIP packages that poke through the board and are soldered to the underside, the leads of SMT IC packages are bent to make flat contact with the top surface of the PCB. Before such components are mounted on the PCB, a special “solder paste” is applied to contact pads on the PCB using a stencil whose hole pattern matches the contact pads to be soldered. Then the SMT components are placed (by hand or by machine) on the pads, where they are held in place by the solder paste (or in some cases, by glue). Finally, the entire assembly is passed through an oven to melt the solder paste, which then solidifies when cooled.

Surface-mount component technology, coupled with fine-line PCB technology, allows extremely dense packing of integrated circuits and other components on a PCB. This dense packing does more than save space. For very high-speed circuits, dense packing goes a long way toward minimizing adverse analog phenomena, including transmission-line effects and speed-of-light limitations.

multichip module (MCM)

To satisfy the most stringent requirements for speed and density, *multichip modules (MCMs)* have been developed. In this technology, IC dice are not mounted in individual plastic or ceramic packages. Instead, the IC dice for a high-speed subsystem (say, a processor and its cache memory) are bonded directly to a substrate that contains the required interconnections on multiple layers. The MCM is hermetically sealed and has its own external pins for power, ground, and just those signals that are required by the system that contains it.

1.10 Digital-Design Levels

Digital design can be carried out at several different levels of representation and abstraction. Although you may learn and practice design at a particular level, from time to time you’ll need to go up or down a level or two to get the job done. Also, the industry itself and most designers have been steadily moving to higher levels of abstraction as circuit density and functionality have increased.

The lowest level of digital design is device physics and IC manufacturing processes. This is the level that is primarily responsible for the breathtaking advances in IC speed and density that have occurred over the past decades. The effects of these advances are summarized in *Moore's Law*, first stated by Intel founder Gordon Moore in 1965: that the number of transistors per square inch in an IC doubles every year. In recent years, the rate of advance has slowed down to doubling about every 18 months, but it is important to note that with each doubling of density has also come a doubling of speed.

Moore's Law

This book does not reach down to the level of device physics and IC processes, but you need to recognize the importance of that level. Being aware of likely technology advances and other changes is important in system and product planning. For example, decreases in chip geometries have recently forced a move to lower logic-power-supply voltages, causing major changes in the way designers plan and specify modular systems and upgrades.

In this book, we jump into digital design at the transistor level and go all the way up to the level of logic design using HDLs. We stop short of the next level, which includes computer design and overall system design. The "center" of our discussion is at the level of functional building blocks.

To get a preview of the levels of design that we'll cover, consider a simple design example. Suppose you are to build a "multiplexer" with two data input bits, A and B, a control input bit S, and an output bit Z. Depending on the value of S, 0 or 1, the circuit is to transfer the value of either A or B to the output Z. This idea is illustrated in the "switch model" of Figure 1-7. Let us consider the design of this function at several different levels.

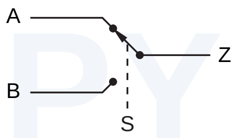


Figure 1-7
Switch model for multiplexer function.

Although logic design is usually carried out at higher level, for some functions it is advantageous to optimize them by designing at the transistor level. The multiplexer is such a function. Figure 1-8 shows how the multiplexer can be designed in "CMOS" technology using specialized transistor circuit structures

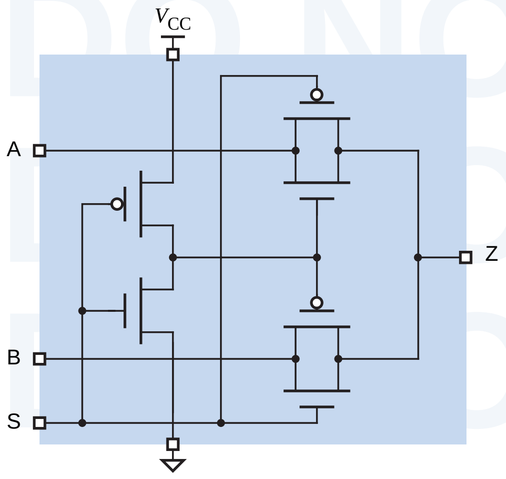


Figure 1-8
Multiplexer design using CMOS transmission gates.

Table 1-1
Truth table for the multiplexer function.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

called "transmission gates," discussed in Section 3.7.1. Using this approach, the multiplexer can be built with just six transistors. Any of the other approaches that we describe require at least 14 transistors.

In the traditional study of logic design, we would use a "truth table" to describe the multiplexer's logic function. A truth table list all possible combinations of input values and the corresponding output values for the function. Since the multiplexer has three inputs, it has 2^3 or 8 possible input combinations, as shown in the truth table in Table 1-1.

Once we have a truth table, traditional logic design methods, described in Section 4.3, use Boolean algebra and well understood minimization algorithms to derive an "optimal" two-level AND-OR equation from the truth table. For the multiplexer truth table, we would derive the following equation:

$$Z = S' \cdot A + S \cdot B$$

This equation is read "Z equals not S and A or S and B." Going one step further, we can convert the equation into a corresponding set of logic gates that perform the specified logic function, as shown in Figure 1-9. This circuit requires 14 transistors if we use standard CMOS technology for the four gates shown.

A multiplexer is a very commonly used function, and most digital logic technologies provide predefined multiplexer building blocks. For example, the 74x157 is an MSI chip that performs multiplexing on two 4-bit inputs simultaneously. Figure 1-10 is a logic diagram that shows how we can hook up just one bit of this 4-bit building block to solve the problem at hand. The numbers in color are pin numbers of a 16-pin DIP package containing the device.

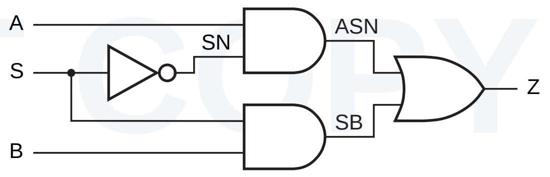


Figure 1-9
Gate-level logic diagram for multiplexer function.

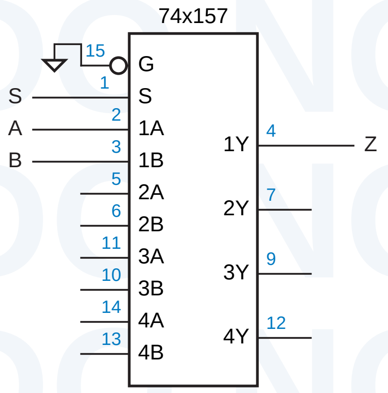


Figure 1-10
Logic diagram for a multiplexer using an MSI building block.

We can also realize the multiplexer function as part of a programmable logic device. Languages like ABEL allow us to specify outputs using Boolean equations similar to the one on the previous page, but it's usually more convenient to use "higher-level" language elements. For example, Table 1-2 is an ABEL program for the multiplexer function. The first three lines define the name of the program module and specify the type of PLD in which the function will be realized. The next two lines specify the device pin numbers for inputs and output. The "`WHEN`" statement specifies the actual logic function in a way that's very easy to understand, even though we haven't covered ABEL yet.

An even higher level language, VHDL, can be used to specify the multiplexer function in a way that is very flexible and hierarchical. Table 1-3 is an example VHDL program for the multiplexer. The first two lines specify a standard library and set of definitions to use in the design. The next four lines specify only the inputs and outputs of the function, and purposely hide any details about the way the function is realized internally. The "architecture" section of the program specifies the function's behavior. VHDL syntax takes a little getting used to, but the single "`when`" statement says basically the same thing that the ABEL version did. A VHDL "synthesis tool" can start with this

Table 1-2
ABEL program for the multiplexer.

```
module chap1mux
title 'Two-input multiplexer example'
CHAP1MUX device 'P16V8'

A, B, S       pin 1, 2, 3;
Z             pin 13 istype 'com';

equations

WHEN S == 0 THEN Z = A;  ELSE Z = B;

end chap1mux
```

Table 1-3
VHDL program for the multiplexer.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vchap1mux is
    port ( A, B, S: in  STD_LOGIC;
           Z:       out STD_LOGIC );
end Vchap1mux;

architecture Vchap1mux_arch of Vchap1mux is
begin
  Z <= A when S = '0' else B;
end Vchap1mux_arch;
```

behavioral description and produce a circuit that has this behavior in a specified target digital-logic technology.

By explicitly enforcing a separation of input/output definitions (“entity”) and internal realization (“architecture”), VHDL makes it easy for designers to define alternate realizations of functions without having to make changes elsewhere in the design hierarchy. For example, a designer could specify an alternate, structural architecture for the multiplexer as shown in Table 1-4. This architecture is basically a text equivalent of the logic diagram in Figure 1-9.

Going one step further, VHDL is powerful enough that we could actually define operations that model functional behavioral at the transistor level (though we won’t explore such capabilities in this book). Thus, we could come full circle by writing a VHDL program that specifies a transistor-level realization of the multiplexer equivalent to Figure 1-8.

Table 1-4
“Structural” VHDL program for the multiplexer.

```
architecture Vchap1mux_gate_arch of Vchap1mux is
signal SN, ASN, SB: STD_LOGIC;
begin
  U1: INV (S, SN);
  U2: AND2 (A, SN, ASN);
  U3: AND2 (S, B, SB);
  U4: OR2 (ASN, SB, Z);
end Vchap1mux_gate_arch;
```

1.11 The Name of the Game

board-level design

Given the functional and performance requirements for a digital system, the name of the game in practical digital design is to minimize cost. For *board-level designs*—systems that are packaged on a single PCB—this usually means minimizing the number of IC packages. If too many ICs are required, they won’t all fit on the PCB. “Well, just use a bigger PCB,” you say. Unfortunately, PCB sizes are usually constrained by factors such as pre-existing standards (e.g., add-in boards for PCs), packaging constraints (e.g., it has to fit in a toaster), or edicts from above (e.g., in order to get the project approved three months ago, you foolishly told your manager that it would all fit on a 3×5 inch PCB, and now you've got to deliver!). In each of these cases, the cost of using a larger PCB or multiple PCBs may be unacceptable.

Minimizing the number of ICs is usually the rule even though individual IC costs vary. For example, a typical SSI or MSI IC may cost 25 cents, while an small PLD may cost a dollar. It may be possible to perform a particular function with three SSI and MSI ICs (75 cents) or one PLD (a dollar). In most situations, the more expensive PLD solution is used, not because the designer owns stock in the IC company, but because the PLD solution uses less PCB area and is also a lot easier to change if it's not right the first time.

ASIC design

In *ASIC design*, the name of the game is a little different, but the importance of structured, functional design techniques is the same. Although it's easy to burn hours and weeks creating custom macrocells and minimizing the total gate count of an ASIC, only rarely is this advisable. The per-unit cost reduction achieved by having a 10% smaller chip is negligible except in high-volume applications. In applications with low to medium volume (the majority), two other factors are more important: design time and NRE cost.

A shorter design time allows a product to reach the market sooner, increasing revenues over the lifetime of the product. A lower NRE cost also flows right to the "bottom line," and in small companies may be the only way the project can be completed before the company runs out of money (believe me, I've been there!). If the product is successful, it's always possible and profitable to "tweak" the design later to reduce per-unit costs. The need to minimize design time and NRE cost argues in favor of a structured, as opposed to highly optimized, approach to ASIC design, using standard building blocks provided in the ASIC manufacturer's library.

The considerations in PLD, CPLD, and FPGA design are a combination of the above. The choice of a particular PLD technology and device size is usually made fairly early in the design cycle. Later, as long as the design "fits" in the selected device, there's no point in trying to optimize gate count or board area—the device has already been committed. However, if new functions or bug fixes push the design beyond the capacity of the selected device, that's when you must work very hard to modify the design to make it fit.

1.12 Going Forward

This concludes the introductory chapter. As you continue reading this book, keep in mind two things. First, the ultimate goal of digital design is to build systems that solve problems for people. While this book will give you the basic tools for design, it's still your job to keep "the big picture" in the back of your mind. Second, cost is an important factor in every design decision; and you must consider not only the cost of digital components, but also the cost of the design activity itself.

Finally, as you get deeper into the text, if you encounter something that you think you've seen before but don't remember where, please consult the index. I've tried to make it as helpful and complete as possible.

Drill Problems

1.1 Suggest some better-looking chapter-opening artwork to put on page 1 of the next edition of this book.

1.2 Give three different definitions for the word "bit" as used in this chapter.

1.3 Define the following acronyms: ASIC, CAD, CD, CO, CPLD, DIP, DVD, FPGA, HDL, IC, IP, LSI, MCM, MSI, NRE, OK, PBX, PCB, PLD, PWB, SMT, SSI, VHDL, VLSI.

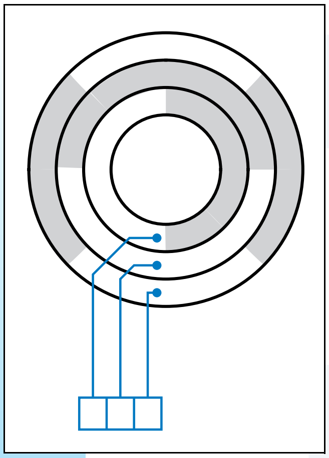
1.4 Research the definitions of the following acronyms: ABEL, CMOS, JPEG, MPEG, OK, PERL, VHDL. (Is OK really an acronym?)

1.5 Excluding the topics in Section 1.2, list three once-analog systems that have "gone digital" since you were born.

1.6 Draw a digital circuit consisting of a 2-input AND gate and three inverters, where an inverter is connected to each of the AND gate's inputs and its output. For each of the four possible combinations of inputs applied to the two primary inputs of this circuit, determine the value produced at the primary output. Is there a simpler circuit that gives the same input/output behavior?

1.7 When should you use the pin diagrams of Figure 1-5 in the schematic diagram of a circuit?

1.8 What is the relationship between "die" and "dice"?



chapter 2

Number Systems and Codes

Digital systems are built from circuits that process binary digits—0s and 1s—yet very few real-life problems are based on binary numbers or any numbers at all. Therefore, a digital system designer must establish some correspondence between the binary digits processed by digital circuits and real-life numbers, events, and conditions. The purpose of this chapter is to show you how familiar numeric quantities can be represented and manipulated in a digital system, and how nonnumeric data, events, and conditions also can be represented.

The first nine sections describe binary number systems and show how addition, subtraction, multiplication, and division are performed in these systems. Sections 2.10–2.13 show how other things, such as decimal numbers, text characters, mechanical positions, and arbitrary conditions, can be encoded using strings of binary digits.

Section 2.14 introduces "*n*-cubes," which provide a way to visualize the relationship between different bit strings. The *n*-cubes are especially useful in the study of error-detecting codes in Section 2.15. We conclude the chapter with an introduction to codes for transmitting and storing data one bit at a time.

2.1 Positional Number Systems

positional number system

weight

The traditional number system that we learned in school and use every day in business is called a *positional number system*. In such a system, a number is represented by a string of digits where each digit position has an associated *weight*. The value of a number is a weighted sum of the digits, for example:

$$1734 = 1 \cdot 1000 + 7 \cdot 100 + 3 \cdot 10 + 4 \cdot 1$$

Each weight is a power of 10 corresponding to the digit's position. A decimal point allows negative as well as positive powers of 10 to be used:

$$5185.68 = 5 \cdot 1000 + 1 \cdot 100 + 8 \cdot 10 + 5 \cdot 1 + 6 \cdot 0.1 + 8 \cdot 0.01$$

In general, a number D of the form $d_1 d_0 . d_{-1} d_{-2}$ has the value

$$D = d_1 \cdot 10^1 + d_0 \cdot 10^0 + d_{-1} \cdot 10^{-1} + d_{-2} \cdot 10^{-2}$$

base

radix

Here, 10 is called the *base* or *radix* of the number system. In a general positional number system, the radix may be any integer $r \geq 2$, and a digit in position i has weight r^i. The general form of a number in such a system is

$$d_{p-1} d_{p-2} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$$

radix point

where there are p digits to the left of the point and n digits to the right of the point, called the *radix point*. If the radix point is missing, it is assumed to be to the right of the rightmost digit. The value of the number is the sum of each digit multiplied by the corresponding power of the radix:

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

high-order digit

most significant digit

low-order digit

least significant digit

Except for possible leading and trailing zeroes, the representation of a number in a positional number system is unique. (Obviously, 0185.6300 equals 185.63, and so on.) The leftmost digit in such a number is called the *high-order* or *most significant digit*; the rightmost is the *low-order* or *least significant digit*.

binary digit

bit

binary radix

As we'll learn in Chapter 3, digital circuits have signals that are normally in one of only two conditions—low or high, charged or discharged, off or on. The signals in these circuits are interpreted to represent *binary digits* (or *bits*) that have one of two values, 0 and 1. Thus, the *binary radix* is normally used to represent numbers in a digital system. The general form of a binary number is

$$b_{p-1} b_{p-2} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$$

and its value is

$$B = \sum_{i=-n}^{p-1} b_i \cdot 2^i$$

In a binary number, the radix point is called the *binary point*. When dealing with binary and other nondecimal numbers, we use a subscript to indicate the radix of each number, unless the radix is clear from the context. Examples of binary numbers and their decimal equivalents are given below.

binary point

$$\begin{aligned} 10011_2 &= 1\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 1\cdot1 = 19_{10} \\ 100010_2 &= 1\cdot32 + 0\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 0\cdot1 = 34_{10} \\ 101.001_2 &= 1\cdot4 + 0\cdot2 + 1\cdot1 + 0\cdot0.5 + 0\cdot0.25 + 1\cdot0.125 = 5.125_{10} \end{aligned}$$

The leftmost bit of a binary number is called the *high-order* or *most significant bit (MSB)*; the rightmost is the *low-order* or *least significant bit (LSB)*.

MSB
LSB

2.2 Octal and Hexadecimal Numbers

Radix 10 is important because we use it in everyday business, and radix 2 is important because binary numbers can be processed directly by digital circuits. Numbers in other radices are not often processed directly, but may be important for documentation or other purposes. In particular, the radices 8 and 16 provide convenient shorthand representations for multibit numbers in a digital system.

The *octal number system* uses radix 8, while the *hexadecimal number system* uses radix 16. Table 2-1 shows the binary integers from 0 to 1111 and their octal, decimal, and hexadecimal equivalents. The octal system needs 8 digits, so it uses digits 0–7 of the decimal system. The hexadecimal system needs 16 digits, so it supplements decimal digits 0–9 with the letters *A–F*.

octal number system
hexadecimal number system
hexadecimal digits A–F

The octal and hexadecimal number systems are useful for representing multibit numbers because their radices are powers of 2. Since a string of three bits can take on eight different combinations, it follows that each 3-bit string can be uniquely represented by one octal digit, according to the third and fourth columns of Table 2-1. Likewise, a 4-bit string can be represented by one hexadecimal digit according to the fifth and sixth columns of the table.

Thus, it is very easy to convert a binary number to octal. Starting at the binary point and working left, we simply separate the bits into groups of three and replace each group with the corresponding octal digit:

binary to octal conversion

$$\begin{aligned} 100011001110_2 &= 100\ 011\ 001\ 110_2 = 4316_8 \\ 11101101110101001_2 &= 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8 \end{aligned}$$

The procedure for binary to hexadecimal conversion is similar, except we use groups of four bits:

binary to hexadecimal conversion

$$\begin{aligned} 100011001110_2 &= 1000\ 1100\ 1110_2 = 8CE_{16} \\ 11101101110101001_2 &= 00011101\ 1011\ 1010\ 1001_2 = 1DBA9_{16} \end{aligned}$$

In these examples we have freely added zeroes on the left to make the total number of bits a multiple of 3 or 4 as required.

Table 2-1 Binary, decimal, octal, and hexadecimal numbers.

Binary	Decimal	Octal	3-Bit String	Hexadecimal	4-Bit String
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

If a binary number contains digits to the right of the binary point, we can convert them to octal or hexadecimal by starting at the binary point and working right. Both the left-hand and right-hand sides can be padded with zeroes to get multiples of three or four bits, as shown in the example below:

$$\begin{aligned} 10.1011001011_2 &= 010\,.\,101\ 100\ 101\ 100_2 = 2.5454_8 \\ &= 0010\,.\,1011\ 0010\ 1100_2 = 2.\mathrm{B2C}_{16} \end{aligned}$$

octal or hexadecimal to binary conversion

Converting in the reverse direction, from octal or hexadecimal to binary, is very easy. We simply replace each octal or hexadecimal digit with the corresponding 3- or 4-bit string, as shown below:

$$\begin{aligned} 1357_8 &= 001\ 011\ 101\ 111_2 \\ 2046.17_8 &= 010\ 000\ 100\ 110\,.\,001\ 111_2 \\ \mathrm{BEAD}_{16} &= 1011\ 1110\ 1010\ 1101_2 \\ 9\mathrm{F}.46\mathrm{C}_{16} &= 1001\ 1111\,.\,0100\ 0110\ 1100_2 \end{aligned}$$

byte

The octal number system was quite popular 25 years ago because of certain minicomputers that had their front-panel lights and switches arranged in groups of three. However, the octal number system is not used much today, because of the preponderance of machines that process 8-bit *bytes*. It is difficult to extract individual byte values in multibyte quantities in the octal representation; for

WHEN I'M 64

As you grow older, you'll find that the hexadecimal number system is useful for more than just computers. When I turned 40, I told friends that I had just turned 28_{16}. The "$_{16}$" was whispered under my breath, of course. At age 50, I'll be only 32_{16}.

People get all excited about decennial birthdays like 20, 30, 40, 50, …, but you should be able to convince your friends that the decimal system is of no fundamental significance. More significant life changes occur around birthdays 2, 4, 8, 16, 32, and 64, when you add a most significant bit to your age. Why do you think the Beatles sang "When I'm sixty-*four*"?

example, what are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

In the hexadecimal system, two digits represent an 8-bit byte, and 2*n* digits represent an *n*-byte word; each pair of digits constitutes exactly one byte. For example, the 32-bit hexadecimal number $5678ABCD_{16}$ consists of four bytes with values 56_{16}, 78_{16}, AB_{16}, and CD_{16}. In this context, a 4-bit hexadecimal digit is sometimes called a *nibble*; a 32-bit (4-byte) number has eight nibbles. Hexadecimal numbers are often used to describe a computer's memory address space. For example, a computer with 16-bit addresses might be described as having read/write memory installed at addresses 0–$EFFF_{16}$, and read-only memory at addresses F000–$FFFF_{16}$. Many computer programming languages use the prefix "0x" to denote a hexadecimal number, for example, `0xBFC0000`.

nibble

`0x` *prefix*

2.3 General Positional Number System Conversions

In general, conversion between two radices cannot be done by simple substitutions; arithmetic operations are required. In this section, we show how to convert a number in any radix to radix 10 and vice versa, using radix-10 arithmetic.

radix-r to decimal conversion

In Section 2.1, we indicated that the value of a number in any radix is given by the formula

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i$$

where *r* is the radix of the number and there are *p* digits to the left of the radix point and *n* to the right. Thus, the value of the number can be found by converting each digit of the number to its radix-10 equivalent and expanding the formula using radix-10 arithmetic. Some examples are given below:

$$\begin{aligned}
1CE8_{16} &= 1\cdot16^3 + 12\cdot16^2 + 14\cdot16^1 + 8\cdot16^0 = 7400_{10} \\
F1A3_{16} &= 15\cdot16^3 + 1\cdot16^2 + 10\cdot16^1 + 3\cdot16^0 = 61859_{10} \\
436.5_8 &= 4\cdot8^2 + 3\cdot8^1 + 6\cdot8^0 + 5\cdot8^{-1} = 286.625_{10} \\
132.3_4 &= 1\cdot4^2 + 3\cdot4^1 + 2\cdot4^0 + 3\cdot4^{-1} = 30.75_{10}
\end{aligned}$$

A shortcut for converting whole numbers to radix 10 is obtained by rewriting the expansion formula as follows:

$$D = ((\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)\cdots r + d_1)\cdot r + d_0$$

That is, we start with a sum of 0; beginning with the leftmost digit, we multiply the sum by r and add the next digit to the sum, repeating until all digits have been processed. For example, we can write

$$\mathrm{F1AC}_{16} = (((15)\cdot 16 + 1\cdot 16 + 10)\cdot 16 + 12$$

decimal to radix-r conversion

Although this formula is not too exciting in itself, it forms the basis for a very convenient method of converting a decimal number D to a radix r. Consider what happens if we divide the formula by r. Since the parenthesized part of the formula is evenly divisible by r, the quotient will be

$$Q = (\cdots((d_{p-1})\cdot r + d_{p-2})\cdot r + \cdots)r + d_1$$

and the remainder will be d_0. Thus, d_0 can be computed as the remainder of the long division of D by r. Furthermore, the quotient Q has the same form as the original formula. Therefore, successive divisions by r will yield successive digits of D from right to left, until all the digits of D have been derived. Examples are given below:

$179 \div 2 = 89$ remainder 1 (LSB)
$\div 2 = 44$ remainder 1
$\div 2 = 22$ remainder 0
$\div 2 = 11$ remainder 0
$\div 2 = 5$ remainder 1
$\div 2 = 2$ remainder 1
$\div 2 = 1$ remainder 0
$\div 2 = 0$ remainder 1 (MSB)
$179_{10} = 10110011_2$

$467 \div 8 = 58$ remainder 3 (least significant digit)
$\div 8 = 7$ remainder 2
$\div 8 = 0$ remainder 7 (most significant digit)
$467_{10} = 723_8$

$3417 \div 16 = 213$ remainder 9 (least significant digit)
$\div 16 = 13$ remainder 5
$\div 16 = 0$ remainder 13 (most significant digit)
$3417_{10} = \mathrm{D59}_{16}$

Table 2-2 summarizes methods for converting among the most common radices.

Table 2-2 Conversion methods for common radices.

Conversion	Method	Example
Binary to		
Octal	Substitution	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Substitution	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Summation	$10111011001_2 = 1 \cdot 1024 + 0 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64$ $+ 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1497_{10}$
Octal to		
Binary	Substitution	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Substitution	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Summation	$1234_8 = 1 \cdot 512 + 2 \cdot 64 + 3 \cdot 8 + 4 \cdot 1 = 668_{10}$
Hexadecimal to		
Binary	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Summation	$C0DE_{16} = 12 \cdot 4096 + 0 \cdot 256 + 13 \cdot 16 + 14 \cdot 1 = 49374_{10}$
Decimal to		
Binary	Division	$108_{10} \div 2 = 54$ remainder 0 (LSB) $\div 2 = 27$ remainder 0 $\div 2 = 13$ remainder 1 $\div 2 = 6$ remainder 1 $\div 2 = 3$ remainder 0 $\div 2 = 1$ remainder 1 $\div 2 = 0$ remainder 1 (MSB) $108_{10} = 1101100_2$
Octal	Division	$108_{10} \div 8 = 13$ remainder 4 (least significant digit) $\div 8 = 1$ remainder 5 $\div 8 = 0$ remainder 1 (most significant digit) $108_{10} = 154_8$
Hexadecimal	Division	$108_{10} \div 16 = 6$ remainder 12 (least significant digit) $\div 16 = 0$ remainder 6 (most significant digit) $108_{10} = 6C_{16}$

Table 2-3
Binary addition and subtraction table.

c_{in} **or** b_{in}	x	y	c_{out}	s	b_{out}	d
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

2.4 Addition and Subtraction of Nondecimal Numbers

Addition and subtraction of nondecimal numbers by hand uses the same technique that we learned in grammar school for decimal numbers; the only catch is that the addition and subtraction tables are different.

binary addition

Table 2-3 is the addition and subtraction table for binary digits. To add two binary numbers X and Y, we add together the least significant bits with an initial carry (c_{in}) of 0, producing carry (c_{out}) and sum (s) bits according to the table. We continue processing bits from right to left, adding the carry out of each column into the next column's sum.

Two examples of decimal additions and the corresponding binary additions are shown in Figure 2-1, using a colored arrow to indicate a carry of 1. The same examples are repeated below along with two more, with the carries shown as a bit string C:

C		101111000	C		001011000
X	190	10111110	X	173	10101101
Y	+141	+ 10001101	Y	+ 44	+ 00101100
$X + Y$	331	101001011	$X + Y$	217	11011001

C		011111110	C		000000000
X	127	01111111	X	170	10101010
Y	+ 63	+ 00111111	Y	+ 85	+ 01010101
$X + Y$	190	10111110	$X + Y$	255	11111111

binary subtraction

minuend

subtrahend

Binary subtraction is performed similarly, using borrows (b_{in} and b_{out}) instead of carries between steps, and producing a difference bit d. Two examples of decimal subtractions and the corresponding binary subtractions are shown in Figure 2-2. As in decimal subtraction, the binary minuend values in the columns are modified when borrows occur, as shown by the colored arrows and bits. The

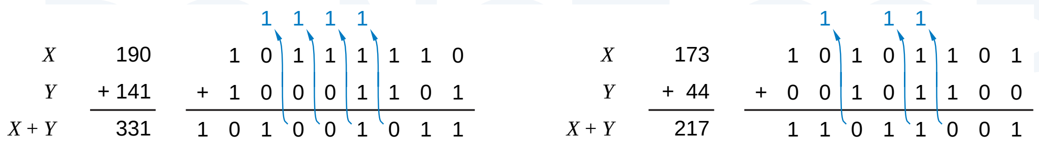


Figure 2-1 Examples of decimal and corresponding binary additions.

examples from the figure are repeated below along with two more, this time showing the borrows as a bit string *B*:

B		001111100
X	229	11100101
Y	− 46	− 00101110
X − *Y*	183	10110111

B		011011010
X	210	11010010
Y	−109	− 01101101
X − *Y*	101	01100101

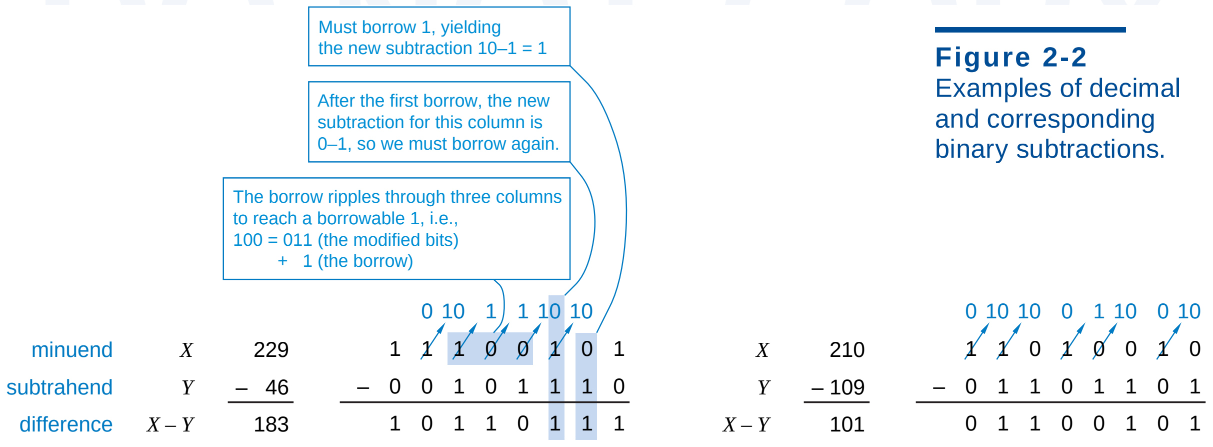
B		010101010
X	170	10101010
Y	− 85	− 01010101
X − *Y*	85	01010101

B		000000000
X	221	11011101
Y	− 76	− 01001100
X − *Y*	145	10010001

comparing numbers

A very common use of subtraction in computers is to compare two numbers. For example, if the operation *X* − *Y* produces a borrow out of the most significant bit position, then *X* is less than *Y*; otherwise, *X* is greater than or equal to *Y*. The relationship between carries and borrow in adders and subtractors will be explored in Section 5.10.

Addition and subtraction tables can be developed for octal and hexadecimal digits, or any other desired radix. However, few computer engineers bother to memorize these tables. If you rarely need to manipulate nondecimal numbers,



Must borrow 1, yielding the new subtraction 10–1 = 1
After the first borrow, the new subtraction for this column is 0–1, so we must borrow again.
The borrow ripples through three columns to reach a borrowable 1, i.e., 100 = 011 (the modified bits) + 1 (the borrow)
0 10 1 1 10 10
minuend X 229 1 1 1 0 0 1 0 1
subtrahend Y − 46 − 0 0 1 0 1 1 1 0
difference X − Y 183 1 0 1 1 0 1 1 1
0 10 10 0 1 10 0 10
X 210 1 1 0 1 0 0 1 0
Y − 109 − 0 1 1 0 1 1 0 1
X − Y 101 0 1 1 0 0 1 0 1

Figure 2-2 Examples of decimal and corresponding binary subtractions.

then it’s easy enough on those occasions to convert them to decimal, calculate results, and convert back. On the other hand, if you must perform calculations in binary, octal, or hexadecimal frequently, then you should ask Santa for a programmer’s “hex calculator” from Texas Instruments or Casio.

If the calculator’s battery wears out, some mental shortcuts can be used to facilitate nondecimal arithmetic. In general, each column addition (or subtraction) can be done by converting the column digits to decimal, adding in decimal, and converting the result to corresponding sum and carry digits in the nondecimal radix. (A carry is produced whenever the column sum equals or exceeds the radix.) Since the addition is done in decimal, we rely on our knowledge of the decimal addition table; the only new thing that we need to learn is the conversion from decimal to nondecimal digits and vice versa. The sequence of steps for mentally adding two hexadecimal numbers is shown below:

hexadecimal addition

C	1 1 0 0	1	1	0	0
X	$1\ 9\ B\ 9_{16}$	1	9	11	9
Y	$+\ C\ 7\ E\ 6_{16}$	+12	7	14	6
X + *Y*	$E\ 1\ 9\ F_{16}$	14	17	25	15
		14	16+1	16+9	15
		E	1	9	F

2.5 Representation of Negative Numbers

So far, we have dealt only with positive numbers, but there are many ways to represent negative numbers. In everyday business, we use the signed-magnitude system, discussed next. However, most computers use one of the complement number systems that we introduce later.

2.5.1 Signed-Magnitude Representation

signed-magnitude system

In the *signed-magnitude system*, a number consists of a magnitude and a symbol indicating whether the magnitude is positive or negative. Thus, we interpret decimal numbers +98, –57, +123.5, and –13 in the usual way, and we also assume that the sign is “+” if no sign symbol is written. There are two possible representations of zero, “+0” and “–0”, but both have the same value.

sign bit

The signed-magnitude system is applied to binary numbers by using an extra bit position to represent the sign (the *sign bit*). Traditionally, the most significant bit (MSB) of a bit string is used as the sign bit (0 = plus, 1 = minus), and the lower-order bits contain the magnitude. Thus, we can write several 8-bit signed-magnitude integers and their decimal equivalents:

$$
\begin{array}{rclcrcl}
01010101_2 & = & +85_{10} & \qquad & 11010101_2 & = & -85_{10} \\
01111111_2 & = & +127_{10} & & 11111111_2 & = & -127_{10} \\
00000000_2 & = & +0_{10} & & 10000000_2 & = & -0_{10}
\end{array}
$$

The signed-magnitude system has an equal number of positive and negative integers. An n-bit signed-magnitude integer lies within the range $-(2^{n-1}-1)$ through $+(2^{n-1}-1)$, and there are two possible representations of zero.

Now suppose that we wanted to build a digital logic circuit that adds signed-magnitude numbers. The circuit must examine the signs of the addends to determine what to do with the magnitudes. If the signs are the same, it must add the magnitudes and give the result the same sign. If the signs are different, it must compare the magnitudes, subtract the smaller from the larger, and give the result the sign of the larger. All of these “ifs,” “adds,” “subtracts,” and “compares” translate into a lot of logic-circuit complexity. Adders for complement number systems are much simpler, as we’ll show next. Perhaps the one redeeming feature of a signed-magnitude system is that, once we know how to build a signed-magnitude adder, a signed-magnitude subtractor is almost trivial to build—it need only change the sign of the subtrahend and pass it along with the minuend to an adder.

signed-magnitude adder

signed-magnitude subtractor

2.5.2 Complement Number Systems

While the signed-magnitude system negates a number by changing its sign, a *complement number system* negates a number by taking its complement as defined by the system. Taking the complement is more difficult than changing the sign, but two numbers in a complement number system can be added or subtracted directly without the sign and magnitude checks required by the signed-magnitude system. We shall describe two complement number systems, called the “radix complement” and the “diminished radix-complement.”

complement number system

In any complement number system, we normally deal with a fixed number of digits, say n. (However, we can increase the number of digits by “sign extension” as shown in Exercise 2.23, and decrease the number by truncating high-order digits as shown in Exercise 2.24.) We further assume that the radix is r, and that numbers have the form

$$D = d_{n-1}d_{n-2}\cdots d_1d_0 .$$

The radix point is on the right and so the number is an integer. If an operation produces a result that requires more than n digits, we throw away the extra high-order digit(s). If a number D is complemented twice, the result is D.

2.5.3 Radix-Complement Representation

In a *radix-complement system*, the complement of an n-digit number is obtained by subtracting it from r^n. In the decimal number system, the radix complement is called the *10’s complement*. Some examples using 4-digit decimal numbers (and subtraction from 10,000) are shown in Table 2-4.

radix-complement system

10’s complement

By definition, the radix complement of an n-digit number D is obtained by subtracting it from r^n. If D is between 1 and $r^n - 1$, this subtraction produces

Table 2-4
Examples of 10's and 9s' complements.

Number	10's complement	9s' complement
1849	8151	8150
2067	7933	7932
100	9900	9899
7	9993	9992
8151	1849	1848
0	10000 (= 0)	9999

another number between 1 and $r^n - 1$. If D is 0, the result of the subtraction is r^n, which has the form $100 \cdots 00$, where there are a total of $n + 1$ digits. We throw away the extra high-order digit and get the result 0. Thus, there is only one representation of zero in a radix-complement system.

computing the radix complement

It seems from the definition that a subtraction operation is needed to compute the radix complement of D. However, this subtraction can be avoided by rewriting r^n as $(r^n - 1) + 1$ and $r^n - D$ as $((r^n - 1) - D) + 1$. The number $r^n - 1$ has the form $mm \cdots mm$, where $m = r - 1$ and there are n m's. For example, 10,000 equals 9,999 + 1. If we define the complement of a digit d to be $r - 1 - d$, then $(r^n - 1) - D$ is obtained by complementing the digits of D. Therefore, the radix complement of a number D is obtained by complementing the individual

Table 2-5
Digit complements.

	Complement			
Digit	Binary	Octal	Decimal	Hexadecimal
0	1	7	9	F
1	0	6	8	E
2	–	5	7	D
3	–	4	6	C
4	–	3	5	B
5	–	2	4	A
6	–	1	3	9
7	–	0	2	8
8	–	–	1	7
9	–	–	0	6
A	–	–	–	5
B	–	–	–	4
C	–	–	–	3
D	–	–	–	2
E	–	–	–	1
F	–	–	–	0

digits of *D* and adding 1. For example, the 10's complement of 1849 is 8150 + 1, or 8151. You should confirm that this trick also works for the other 10's-complement examples above. Table 2-5 lists the digit complements for binary, octal, decimal, and hexadecimal numbers.

2.5.4 Two's-Complement Representation

two's complement

For binary numbers, the radix complement is called the *two's complement*. The MSB of a number in this system serves as the sign bit; a number is negative if and only if its MSB is 1. The decimal equivalent for a two's-complement binary number is computed the same way as for an unsigned number, except that the weight of the MSB is -2^{n-1} instead of $+2^{n-1}$. The range of representable numbers is $-(2^{n-1})$ through $+(2^{n-1}-1)$. Some 8-bit examples are shown below:

weight of MSB

$$\begin{array}{rrl} 17_{10} = & 00010001_2 & \\ & \Downarrow & \text{complement bits} \\ & 11101110 & \\ & +1 & \\ \hline & 11101111_2 & = -17_{10} \end{array} \qquad \begin{array}{rrl} -99_{10} = & 10011101_2 & \\ & \Downarrow & \text{complement bits} \\ & 01100010 & \\ & +1 & \\ \hline & 01100011_2 & = 99_{10} \end{array}$$

$$\begin{array}{rrl} 119_{10} = & 01110111 & \\ & \Downarrow & \text{complement bits} \\ & 10001000 & \\ & +1 & \\ \hline & 10001001_2 & = -119_{10} \end{array} \qquad \begin{array}{rrl} -127_{10} = & 10000001 & \\ & \Downarrow & \text{complement bits} \\ & 01111110 & \\ & +1 & \\ \hline & 01111111_2 & = 127_{10} \end{array}$$

$$\begin{array}{rrl} 0_{10} = & 00000000_2 & \\ & \Downarrow & \text{complement bits} \\ & 11111111 & \\ & +1 & \\ \hline & 1\ 00000000_2 & = 0_{10} \end{array} \qquad \begin{array}{rrl} -128_{10} = & 10000000_2 & \\ & \Downarrow & \text{complement bits} \\ & 01111111 & \\ & +1 & \\ \hline & 10000000_2 & = -128_{10} \end{array}$$

A carry out of the MSB position occurs in one case, as shown in color above. As in all two's-complement operations, this bit is ignored and only the low-order *n* bits of the result are used.

In the two's-complement number system, zero is considered positive because its sign bit is 0. Since two's complement has only one representation of zero, we end up with one extra negative number, $-(2^{n-1})$, that doesn't have a positive counterpart.

extra negative number

We can convert an *n*-bit two's-complement number *X* into an *m*-bit one, but some care is needed. If $m > n$, we must append $m - n$ copies of *X*'s sign bit to the left of *X* (see Exercise 2.23). That is, we pad a positive number with 0s and a negative one with 1s; this is called *sign extension*. If $m < n$, we discard *X*'s $n - m$

sign extension

leftmost bits; however, the result is valid only if all of the discarded bits are the same as the sign bit of the result (see Exercise 2.24).

Most computers and other digital systems use the two’s-complement system to represent negative numbers. However, for completeness, we’ll also describe the diminished radix-complement and ones’-complement systems.

*2.5.5 Diminished Radix-Complement Representation

diminished radix-complement system

9s’ complement

In a *diminished radix-complement system*, the complement of an *n*-digit number *D* is obtained by subtracting it from r^n-1. This can be accomplished by complementing the individual digits of *D*, *without* adding 1 as in the radix-complement system. In decimal, this is called the *9s’ complement*; some examples are given in the last column of Table 2-4 on page 32.

*2.5.6 Ones’-Complement Representation

ones’ complement

The diminished radix-complement system for binary numbers is called the *ones’ complement*. As in two’s complement, the most significant bit is the sign, 0 if positive and 1 if negative. Thus there are two representations of zero, positive zero ($00\cdots00$) and negative zero ($11\cdots11$). Positive number representations are the same for both ones’ and two’s complements. However, negative number representations differ by 1. A weight of $-(2^{n-1}-1)$, rather than -2^{n-1}, is given to the most significant bit when computing the decimal equivalent of a ones’-complement number. The range of representable numbers is $-(2^{n-1}-1)$ through $+(2^{n-1}-1)$. Some 8-bit numbers and their ones’ complements are shown below:

$$17_{10} = 00010001_2 \quad \Downarrow \quad 11101110_2 = -17_{10}$$

$$-99_{10} = 10011100_2 \quad \Downarrow \quad 01100011_2 = 99_{10}$$

$$119_{10} = 01110111_2 \quad \Downarrow \quad 10001000_2 = -119_{10}$$

$$-127_{10} = 10000000_2 \quad \Downarrow \quad 01111111_2 = 127_{10}$$

$$0_{10} = 00000000_2 \text{ (positive zero)} \quad \Downarrow \quad 11111111_2 = 0_{10} \text{ (negative zero)}$$

The main advantages of the ones’-complement system are its symmetry and the ease of complementation. However, the adder design for ones’-complement numbers is somewhat trickier than a two’s-complement adder (see Exercise 7.67). Also, zero-detecting circuits in a ones’-complement system

* Throughout this book, *optional sections* are marked with an asterisk.

either must check for both representations of zero, or must always convert $11\cdots11$ to $00\cdots00$.

*2.5.7 Excess Representations

Yes, the number of different systems for representing negative numbers *is* excessive, but there's just one more for us to cover. In *excess-B representation*, an m-bit string whose unsigned integer value is M ($0 \le M < 2^m$) represents the signed integer $M - B$, where B is called the *bias* of the number system.

excess-B representation

bias

excess-2^{m-1} system

For example, an *excess–2^{m-1} system* represents any number X in the range -2^{m-1} through $+2^{m-1} - 1$ by the m-bit binary representation of $X + 2^{m-1}$ (which is always nonnegative and less than 2^m). The range of this representation is exactly the same as that of m-bit two's-complement numbers. In fact, the representations of any number in the two systems are identical except for the sign bits, which are always opposite. (Note that this is true only when the bias is 2^{m-1}.)

The most common use of excess representations is in floating-point number systems (see References).

2.6 Two's-Complement Addition and Subtraction

2.6.1 Addition Rules

A table of decimal numbers and their equivalents in different number systems, Table 2-6, reveals why the two's complement is preferred for arithmetic operations. If we start with 1000_2 (-8_{10}) and count up, we see that each successive two's-complement number all the way to 0111_2 ($+7_{10}$) can be obtained by adding 1 to the previous one, ignoring any carries beyond the fourth bit position. The same cannot be said of signed-magnitude and ones'-complement numbers. Because ordinary addition is just an extension of counting, two's-complement numbers can thus be added by ordinary binary addition, ignoring any carries beyond the MSB. The result will always be the correct sum as long as the range of the number system is not exceeded. Some examples of decimal addition and the corresponding 4-bit two's-complement additions confirm this:

two's-complement addition

$$\begin{array}{rr} +3 & 0011 \\ + \;\; +4 & + \;\; 0100 \\ \hline +7 & 0111 \end{array} \qquad \begin{array}{rr} -2 & 1110 \\ + \;\; -6 & + \;\; 1010 \\ \hline -8 & 11000 \end{array}$$

$$\begin{array}{rr} +6 & 0110 \\ + \;\; -3 & + \;\; 1101 \\ \hline +3 & 10011 \end{array} \qquad \begin{array}{rr} +4 & 0100 \\ + \;\; -7 & + \;\; 1001 \\ \hline -3 & 1101 \end{array}$$

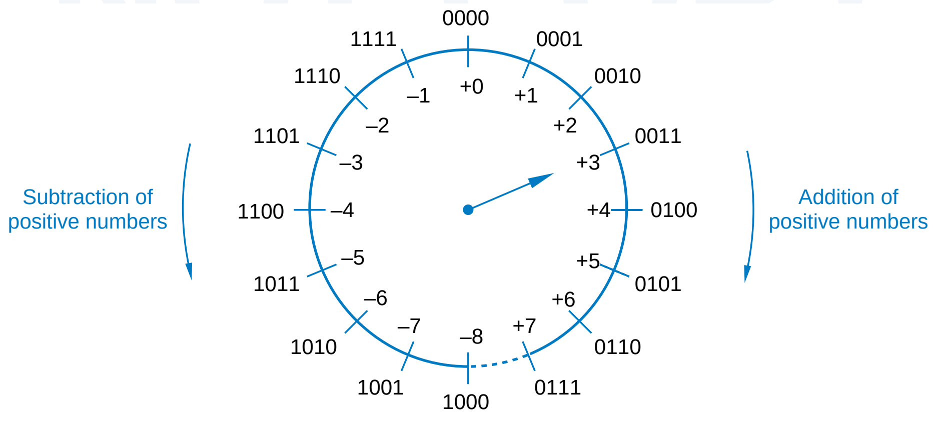
Table 2-6 Decimal and 4-bit numbers.

Decimal	*Two's Complement*	*Ones' Complement*	*Signed Magnitude*	*Excess 2^{m-1}*
–8	1000	—	—	0000
–7	1001	1000	1111	0001
–6	1010	1001	1110	0010
–5	1011	1010	1101	0011
–4	1100	1011	1100	0100
–3	1101	1100	1011	0101
–2	1110	1101	1010	0110
–1	1111	1110	1001	0111
0	0000	1111 or 0000	1000 or 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

2.6.2 A Graphical View

Another way to view the two's-complement system uses the 4-bit "counter" shown in Figure 2-3. Here we have shown the numbers in a circular or "modular" representation. The operation of this counter very closely mimics that of a real up/down counter circuit, which we'll study in Section 8.4. Starting

Figure 2-3 A modular counting representation of 4-bit two's-complement numbers.



with the arrow pointing to any number, we can add $+n$ to that number by counting up n times, that is, by moving the arrow n positions clockwise. It is also evident that we can subtract n from a number by counting down n times, that is, by moving the arrow n positions counterclockwise. Of course, these operations give correct results only if n is small enough that we don't cross the discontinuity between −8 and +7.

What is most interesting is that we can also subtract n (or add $-n$) by moving the arrow $16 - n$ positions clockwise. Notice that the quantity $16 - n$ is what we defined to be the 4-bit two's complement of n, that is, the two's-complement representation of $-n$. This graphically supports our earlier claim that a negative number in two's-complement representation may be added to another number simply by adding the 4-bit representations using ordinary binary addition. Adding a number in Figure 2-3 is equivalent to moving the arrow a corresponding number of positions clockwise.

2.6.3 Overflow

overflow

If an addition operation produces a result that exceeds the range of the number system, *overflow* is said to occur. In the modular counting representation of Figure 2-3, overflow occurs during addition of positive numbers when we count past +7. Addition of two numbers with different signs can never produce overflow, but addition of two numbers of like sign can, as shown by the following examples:

$$\begin{array}{rr} & -3 \\ + & -6 \\ \hline & -9 \end{array} \quad \begin{array}{rr} & 1101 \\ + & 1010 \\ \hline & 10111 \end{array} = +7 \qquad \begin{array}{rr} & +5 \\ + & +6 \\ \hline & +11 \end{array} \quad \begin{array}{rr} & 0101 \\ + & 0110 \\ \hline & 1011 \end{array} = -5$$

$$\begin{array}{rr} & -8 \\ + & -8 \\ \hline & -16 \end{array} \quad \begin{array}{rr} & 1000 \\ + & 1000 \\ \hline & 10000 \end{array} = +0 \qquad \begin{array}{rr} & +7 \\ + & +7 \\ \hline & +14 \end{array} \quad \begin{array}{rr} & 0111 \\ + & 0111 \\ \hline & 1110 \end{array} = -2$$

overflow rules

Fortunately, there is a simple rule for detecting overflow in addition: An addition overflows if the signs of the addends are the same and the sign of the sum is different from the addends' sign. The overflow rule is sometimes stated in terms of carries generated during the addition operation: An addition overflows if the carry bits c_{in} into and c_{out} out of the sign position are different. Close examination of Table 2-3 on page 28 shows that the two rules are equivalent—there are only two cases where $c_{\text{in}} \neq c_{\text{out}}$, and these are the only two cases where $x = y$ and the sum bit is different.

2.6.4 Subtraction Rules

two's-complement subtraction

Two's-complement numbers may be subtracted as if they were ordinary unsigned binary numbers, and appropriate rules for detecting overflow may be formulated. However, most subtraction circuits for two's-complement numbers do not perform subtraction directly. Rather, they negate the subtrahend by taking its two's complement, and then add it to the minuend using the normal rules for addition.

Negating the subtrahend and adding the minuend can be accomplished with only one addition operation as follows: Perform a bit-by-bit complement of the subtrahend and add the complemented subtrahend to the minuend with an initial carry (c_{in}) of 1 instead of 0. Examples are given below:

$$\begin{array}{rrrl}
 & & 1 & — c_{\text{in}} \\
+4 & 0100 & 0100 & \\
-\;+3 & -\;0011 & +\;1100 & \\
\hline
+3 & & 10001 &
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & — c_{\text{in}} \\
+3 & 0011 & 0011 & \\
-\;+4 & -\;0100 & +\;1011 & \\
\hline
-1 & & 1111 &
\end{array}$$

$$\begin{array}{rrrl}
 & & 1 & — c_{\text{in}} \\
+3 & 0011 & 0011 & \\
-\;-4 & -\;1100 & +\;0011 & \\
\hline
+7 & & 0111 &
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & — c_{\text{in}} \\
-3 & 1101 & 1101 & \\
-\;-4 & -\;1100 & +\;0011 & \\
\hline
+1 & & 10001 &
\end{array}$$

Overflow in subtraction can be detected by examining the signs of the minuend and the *complemented* subtrahend, using the same rule as in addition. Or, using the technique in the preceding examples, the carries into and out of the sign position can be observed and overflow detected irrespective of the signs of inputs and output, again using the same rule as in addition.

An attempt to negate the "extra" negative number results in overflow according to the rules above, when we add 1 in the complementation process:

$$\begin{array}{rrl}
-(-8) = -1000 = & 0111 & \\
 & +\;0001 & \\
\hline
 & 1000 & = -8
\end{array}$$

However, this number can still be used in additions and subtractions as long as the final result does not exceed the number range:

$$\begin{array}{rr}
+4 & 0100 \\
+\;-8 & +\;1000 \\
\hline
-4 & 1100
\end{array}
\qquad
\begin{array}{rrrl}
 & & 1 & — c_{\text{in}} \\
-3 & 1101 & 1101 & \\
-\;-8 & -\;1000 & +\;0111 & \\
\hline
+5 & & 10101 &
\end{array}$$

2.6.5 Two's-Complement and Unsigned Binary Numbers

Since two's-complement numbers are added and subtracted by the same basic binary addition and subtraction algorithms as unsigned numbers of the same length, a computer or other digital system can use the same adder circuit to handle numbers of both types. However, the results must be interpreted differently depending on whether the system is dealing with signed numbers (e.g., −8 through +7) or unsigned numbers (e.g., 0 through 15).

signed vs. unsigned numbers

We introduced a graphical representation of the 4-bit two's-complement system in Figure 2-3. We can relabel this figure as shown in Figure 2-4 to obtain a representation of the 4-bit unsigned numbers. The binary combinations occupy the same positions on the wheel, and a number is still added by moving the arrow a corresponding number of positions clockwise, and subtracted by moving the arrow counterclockwise.

An addition operation can be seen to exceed the range of the 4-bit unsigned number system in Figure 2-4 if the arrow moves clockwise through the discontinuity between 0 and 15. In this case a *carry* out of the most significant bit position is said to occur.

carry

Likewise a subtraction operation exceeds the range of the number system if the arrow moves counterclockwise through the discontinuity. In this case a *borrow* out of the most significant bit position is said to occur.

borrow

From Figure 2-4 it is also evident that we may subtract an unsigned number n by counting *clockwise* $16 - n$ positions. This is equivalent to *adding* the 4-bit two's-complement of n. The subtraction produces a borrow if the corresponding addition of the two's complement *does not* produce a carry.

In summary, in unsigned addition the carry or borrow in the most significant bit position indicates an out-of-range result. In signed, two's-complement addition the overflow condition defined earlier indicates an out-of-range result. The carry from the most significant bit position is irrelevant in signed addition in the sense that overflow may or may not occur independently of whether or not a carry occurs.

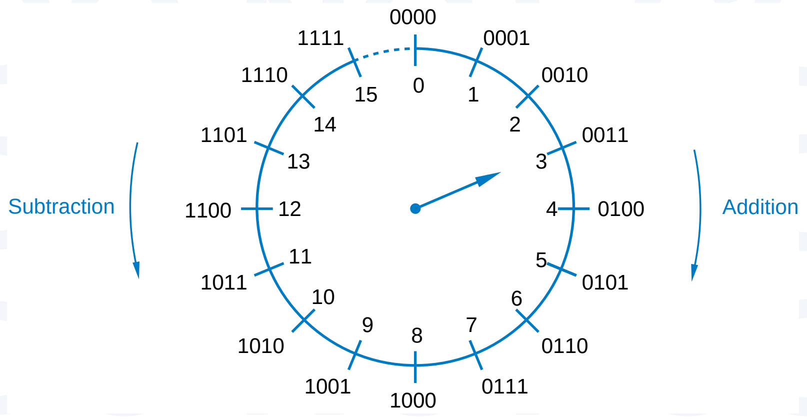


Figure 2-4
A modular counting representation of 4-bit unsigned numbers.

*2.7 Ones'-Complement Addition and Subtraction

Another look at Table 2-6 helps to explain the rule for adding ones'-complement numbers. If we start at 1000_2 (-7_{10}) and count up, we obtain each successive ones'-complement number by adding 1 to the previous one, *except* at the transition from 1111_2 (negative 0) to 0001_2 ($+1_{10}$). To maintain the proper count, we must add 2 instead of 1 whenever we count past 1111_2. This suggests a technique for adding ones'-complement numbers: Perform a standard binary addition, but add an extra 1 whenever we count past 1111_2.

ones'-complement addition

Counting past 1111_2 during an addition can be detected by observing the carry out of the sign position. Thus, the rule for adding ones'-complement numbers can be stated quite simply:

- Perform a standard binary addition; if there is a carry out of the sign position, add 1 to the result.

end-around carry

This rule is often called *end-around carry*. Examples of ones'-complement addition are given below; the last three include an end-around carry:

$$\begin{array}{rrcrrcrr}
+3 & 0011 & \qquad & +4 & 0100 & \qquad & +5 & 0101 \\
+\ {+4} & +\ 0100 & & +\ {-7} & +\ 1000 & & +\ {-5} & +\ 1010 \\ \cline{1-2}\cline{4-5}\cline{7-8}
+7 & 0111 & & -3 & 1100 & & -0 & 1111
\end{array}$$

$$\begin{array}{rrcrrcrr}
-2 & 1101 & \qquad & +6 & 0110 & \qquad & -0 & 1111 \\
+\ {-5} & +\ 1010 & & +\ {-3} & +\ 1100 & & +\ {-0} & +\ 1111 \\ \cline{1-2}\cline{4-5}\cline{7-8}
-7 & 10111 & & +3 & 10010 & & -0 & 11110 \\
 & +\ \quad 1 & & & +\ \quad 1 & & & +\ \quad 1 \\ \cline{2-2}\cline{5-5}\cline{8-8}
 & 1000 & & & 0011 & & & 1111
\end{array}$$

Following the two-step addition rule above, the addition of a number and its ones' complement produces negative 0. In fact, an addition operation using this rule can never produce positive 0 unless both addends are positive 0.

ones'-complement subtraction

As with two's complement, the easiest way to do ones'-complement subtraction is to complement the subtrahend and add. Overflow rules for ones'-complement addition and subtraction are the same as for two's complement.

Table 2-7 summarizes the rules that we presented in this and previous sections for negation, addition, and subtraction in binary number systems.

Table 2-7 Summary of addition and subtraction rules for binary numbers.

Number System	*Addition Rules*	*Negation Rules*	*Subtraction Rules*
Unsigned	Add the numbers. Result is out of range if a carry out of the MSB occurs.	Not applicable	Subtract the subtrahend from the minuend. Result is out of range if a borrow out of the MSB occurs.
Signed magnitude	(same sign) Add the magnitudes; overflow occurs if a carry out of MSB occurs; result has the same sign. (opposite sign) Subtract the smaller magnitude from the larger; overflow is impossible; result has the sign of the larger.	Change the number's sign bit.	Change the sign bit of the subtrahend and proceed as in addition.
Two's complement	Add, ignoring any carry out of the MSB. Overflow occurs if the carries into and out of MSB are different.	Complement all bits of the number; add 1 to the result.	Complement all bits of the subtrahend and add to the minuend with an initial carry of 1.
Ones' complement	Add; if there is a carry out of the MSB, add 1 to the result. Overflow if carries into and out of MSB are different.	Complement all bits of the number.	Complement all bits of the subtrahend and proceed as in addition.

*2.8 Binary Multiplication

shift-and-add multiplication

unsigned binary multiplication

In grammar school we learned to multiply by adding a list of shifted multiplicands computed according to the digits of the multiplier. The same method can be used to obtain the product of two unsigned binary numbers. Forming the shifted multiplicands is trivial in binary multiplication, since the only possible values of the multiplier digits are 0 and 1. An example is shown below:

$$\begin{array}{rr} & 11\\ \times & 13\\ \hline & 33\\ 1 & 1\\ \hline 1 & 43 \end{array} \qquad \begin{array}{rl} 1011 & \text{multiplicand}\\ \times\quad 1101 & \text{multiplier}\\ \hline 1011 & \\ 0000 & \left.\vphantom{\begin{array}{c}1\\1\\1\\1\end{array}}\right\}\text{shifted multiplicands}\\ 1011 & \\ 1011 & \\ \hline 10001111 & \text{product} \end{array}$$

partial product

Instead of listing all the shifted multiplicands and then adding, in a digital system it is more convenient to add each shifted multiplicand as it is created to a *partial product*. Applying this technique to the previous example, four additions and partial products are used to multiply 4-bit numbers:

```
     11            1011   multiplicand
   × 13      ×     1101   multiplier
   ----      ----------
                   0000   partial product
                   1011   shifted multiplicand
             ----------
                  01011   partial product
                  0000↓   shifted multiplicand
             ----------
                 001011   partial product
                 1011↓↓   shifted multiplicand
             ----------
                0110111   partial product
                1011↓↓↓   shifted multiplicand
             ----------
               10001111   product
```

In general, when we multiply an n-bit number by an m-bit number, the resulting product requires at most $n + m$ bits to express. The shift-and-add algorithm requires m partial products and additions to obtain the result, but the first addition is trivial, since the first partial product is zero. Although the first partial product has only n significant bits, after each addition step the partial product gains one more significant bit, since each addition may produce a carry. At the same time, each step yields one more partial product bit, starting with the rightmost and working toward the left, that does not change. The shift-and-add algorithm can be performed by a digital circuit that includes a shift register, an adder, and control logic, as shown in Section 8.7.2.

signed multiplication

Multiplication of signed numbers can be accomplished using unsigned multiplication and the usual grammar school rules: Perform an unsigned multiplication of the magnitudes and make the product positive if the operands had the same sign, negative if they had different signs. This is very convenient in signed-magnitude systems, since the sign and magnitude are separate.

two's-complement multiplication

In the two's-complement system, obtaining the magnitude of a negative number and negating the unsigned product are nontrivial operations. This leads us to seek a more efficient way of performing two's-complement multiplication, described next.

Conceptually, unsigned multiplication is accomplished by a sequence of unsigned additions of the shifted multiplicands; at each step, the shift of the multiplicand corresponds to the weight of the multiplier bit. The bits in a two's-complement number have the same weights as in an unsigned number, except for the MSB, which has a negative weight (see Section 2.5.4). Thus, we can perform two's-complement multiplication by a sequence of two's-complement additions of shifted multiplicands, except for the last step, in which the shifted multiplicand corresponding to the MSB of the multiplier must be negated before it is added to the partial product. Our previous example is repeated below, this time interpreting the multiplier and multiplicand as two's-complement numbers:

−5	1011	multiplicand
× −3	× 1101	multiplier
	00000	partial product
	11011	shifted multiplicand
	111011	partial product
	00000↓	shifted multiplicand
	1111011	partial product
	11011↓↓	shifted multiplicand
	11100111	partial product
	00101↓↓↓	shifted and negated multiplicand
	00001111	product

Handling the MSBs is a little tricky because we gain one significant bit at each step and we are working with signed numbers. Therefore, before adding each shifted multiplicand and k-bit partial product, we change them to $k + 1$ significant bits by sign extension, as shown in color above. Each resulting sum has $k + 1$ bits; any carry out of the MSB of the $k + 1$-bit sum is ignored.

*2.9 Binary Division

shift-and-subtract division

unsigned division

The simplest binary division algorithm is based on the shift-and-subtract method that we learned in grammar school. Table 2-8 gives examples of this method for unsigned decimal and binary numbers. In both cases, we mentally compare the

Table 2-8 Example of long division.

19	10011	quotient
11)217	1011)11011001	dividend
11	1011	shifted divisor
107	0101	reduced dividend
99	0000	shifted divisor
8	1010	reduced dividend
	0000	shifted divisor
	10100	reduced dividend
	1011	shifted divisor
	10011	reduced dividend
	1011	shifted divisor
	1000	remainder

reduced dividend with multiples of the divisor to determine which multiple of the shifted divisor to subtract. In the decimal case, we first pick 11 as the greatest multiple of 11 less than 21, and then pick 99 as the greatest multiple less than 107. In the binary case, the choice is somewhat simpler, since the only two choices are zero and the divisor itself.

Division methods for binary numbers are somewhat complementary to binary multiplication methods. A typical division algorithm accepts an n+m-bit dividend and an n-bit divisor, and produces an m-bit quotient and an n-bit remainder. A division *overflows* if the divisor is zero or the quotient would take more than m bits to express. In most computer division circuits, $n = m$.

division overflow

signed division

Division of signed numbers can be accomplished using unsigned division and the usual grammar school rules: Perform an unsigned division of the magnitudes and make the quotient positive if the operands had the same sign, negative if they had different signs. The remainder should be given the same sign as the dividend. As in multiplication, there are special techniques for performing division directly on two's-complement numbers; these techniques are often implemented in computer division circuits (see References).

2.10 Binary Codes for Decimal Numbers

Even though binary numbers are the most appropriate for the internal computations of a digital system, most people still prefer to deal with decimal numbers. As a result, the external interfaces of a digital system may read or display decimal numbers, and some digital devices actually process decimal numbers directly.

The human need to represent decimal numbers doesn't change the basic nature of digital electronic circuits—they still process signals that take on one of only two states that we call 0 and 1. Therefore, a decimal number is represented in a digital system by a string of bits, where different combinations of bit values in the string represent different decimal numbers. For example, if we use a 4-bit string to represent a decimal number, we might assign bit combination 0000 to decimal digit 0, 0001 to 1, 0010 to 2, and so on.

code

code word

A set of n-bit strings in which different bit strings represent different numbers or other things is called a *code*. A particular combination of n bit-values is called a *code word*. As we'll see in the examples of decimal codes in this section, there may or may not be an arithmetic relationship between the bit values in a code word and the thing that it represents. Furthermore, a code that uses n-bit strings need not contain 2^n valid code words.

At least four bits are needed to represent the ten decimal digits. There are billions and billions of different ways to choose ten 4-bit code words, but some of the more common decimal codes are listed in Table 2-9.

binary-coded decimal (BCD)

Perhaps the most "natural" decimal code is *binary-coded decimal (BCD)*, which encodes the digits 0 through 9 by their 4-bit unsigned binary representa-

Table 2-9 Decimal codes.

Decimal digit	BCD (8421)	2421	Excess-3	Biquinary	1-out-of-10
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
			Unused code words		
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	. . .	. . .

tions, 0000 through 1001. The code words 1010 through 1111 are not used. Conversions between BCD and decimal representations are trivial, a direct substitution of four bits for each decimal digit. Some computer programs place two BCD digits in one 8-bit byte in *packed-BCD representation*; thus, one byte may represent the values from 0 to 99 as opposed to 0 to 255 for a normal unsigned 8-bit binary number. BCD numbers with any desired number of digits may be obtained by using one byte for each two digits.

packed-BCD representation

As with binary numbers, there are many possible representations of negative BCD numbers. Signed BCD numbers have one extra digit position for the

BINOMIAL COEFFICIENTS

The number of different ways to choose m items from a set of n items is given by a *binomial coefficient*, denoted $\binom{n}{m}$, whose value is $\frac{n!}{m! \cdot (n-m!)}$. For a 4-bit decimal code, there are $\binom{16}{10}$ different ways to choose 10 out of 16 4-bit code words, and 10! ways to assign each different choice to the 10 digits. So there are $\frac{16!}{10! \cdot 6!} \cdot 10!$ or 29,059,430,400 different 4-bit decimal codes.

sign. Both the signed-magnitude and 10’s-complement representations are popular. In signed-magnitude BCD, the encoding of the sign bit string is arbitrary; in 10’s-complement, 0000 indicates plus and 1001 indicates minus.

BCD addition

Addition of BCD digits is similar to adding 4-bit unsigned binary numbers, except that a correction must be made if a result exceeds 1001. The result is corrected by adding 6; examples are shown below:

```
     5        0101                          4        0100
 +   9    +   1001                      +   5    +   0101
 -----    --------                      -----    --------
    14        1110                          9        1001
          +   0110 — correction
          --------
  10+4      1 0100

     8        1000                          9        1001
 +   8    +   1000                        + 9    +   1001
 -----    --------                      -----    --------
   –16      1 0000                         18      1 0010
          +   0110 — correction                  +   0110 — correction
          --------                               --------
  10+6      1 0110                       10+8      1 1000
```

Notice that the addition of two BCD digits produces a carry into the next digit position if either the initial binary addition or the correction factor addition produces a carry. Many computers perform packed-BCD arithmetic using special instructions that handle the carry correction automatically.

weighted code

8421 code

2421 code

self-complementing code

Binary-coded decimal is a *weighted code* because each decimal digit can be obtained from its code word by assigning a fixed weight to each code-word bit. The weights for the BCD bits are 8, 4, 2, and 1, and for this reason the code is sometimes called the *8421 code*. Another set of weights results in the *2421 code* shown in Table 2-9. This code has the advantage that it is *self-complementing*, that is, the code word for the 9s’ complement of any digit may be obtained by complementing the individual bits of the digit’s code word.

excess-3 code

Another self-complementing code shown in Table 2-9 is the *excess-3 code*. Although this code is not weighted, it has an arithmetic relationship with the BCD code—the code word for each decimal digit is the corresponding BCD code word plus 0011_2. Because the code words follow a standard binary counting sequence, standard binary counters can easily be made to count in excess-3 code, as we’ll show in Figure 8-37 on page 600.

biquinary code

Decimal codes can have more than four bits; for example, the *biquinary code* in Table 2-9 uses seven. The first two bits in a code word indicate whether the number is in the range 0–4 or 5–9, and the last five bits indicate which of the five numbers in the selected range is represented.

One potential advantage of using more than the minimum number of bits in a code is an error-detecting property. In the biquinary code, if any one bit in a code word is accidentally changed to the opposite value, the resulting code word does not represent a decimal digit and can therefore be flagged as an error. Out of 128 possible 7-bit code words, only 10 are valid and recognized as decimal digits; the rest can be flagged as errors if they appear.

1-out-of-10 code

A *1-out-of-10 code* such as the one shown in the last column of Table 2-9 is the sparsest encoding for decimal digits, using 10 out of 1024 possible 10-bit code words.

2.11 Gray Code

In electromechanical applications of digital systems—such as machine tools, automotive braking systems, and copiers—it is sometimes necessary for an input sensor to produce a digital value that indicates a mechanical position. For example, Figure 2-5 is a conceptual sketch of an encoding disk and a set of contacts that produce one of eight 3-bit binary-coded values depending on the rotational position of the disk. The dark areas of the disk are connected to a signal source corresponding to logic 1, and the light areas are unconnected, which the contacts interpret as logic 0.

The encoder in Figure 2-5 has a problem when the disk is positioned at certain boundaries between the regions. For example, consider the boundary between the 001 and 010 regions of the disk; two of the encoded bits change here. What value will the encoder produce if the disk is positioned right on the theoretical boundary? Since we're on the border, both 001 and 010 are acceptable. However, because the mechanical assembly is not perfect, the two right-hand contacts may both touch a "1" region, giving an incorrect reading of 011. Likewise, a reading of 000 is possible. In general, this sort of problem can occur at any boundary where more than one bit changes. The worst problems occur when all three bits are changing, as at the 000–111 and 011–100 boundaries.

Gray code

The encoding-disk problem can be solved by devising a digital code in which only one bit changes between each pair of successive code words. Such a code is called a *Gray code*; a 3-bit Gray code is listed in Table 2-10. We've rede-

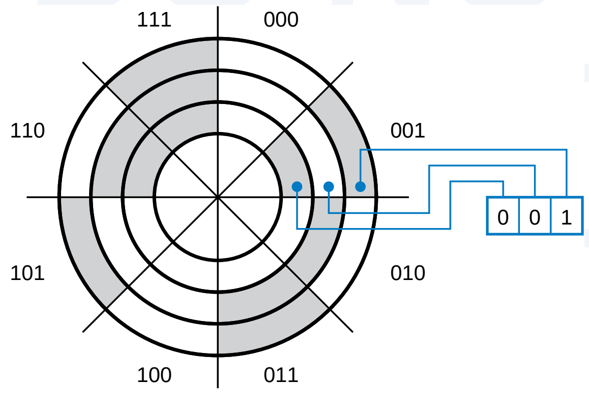


Figure 2-5
A mechanical encoding disk using a 3-bit binary code.

Table 2-10
A comparison of 3-bit binary code and Gray code.

Decimal number	*Binary code*	*Gray code*
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

signed the encoding disk using this code as shown in Figure 2-6. Only one bit of the new disk changes at each border, so borderline readings give us a value on one side or the other of the border.

reflected code

There are two convenient ways to construct a Gray code with any desired number of bits. The first method is based on the fact that Gray code is a *reflected code*; it can be defined (and constructed) recursively using the following rules:

1. A 1-bit Gray code has two code words, 0 and 1.
2. The first 2^n code words of an n+1-bit Gray code equal the code words of an n-bit Gray code, written in order with a leading 0 appended.
3. The last 2^n code words of an n+1-bit Gray code equal the code words of an n-bit Gray code, but written in reverse order with a leading 1 appended.

If we draw a line between rows 3 and 4 of Table 2-10, we can see that rules 2 and 3 are true for the 3-bit Gray code. Of course, to construct an n-bit Gray code for an arbitrary value of n with this method, we must also construct a Gray code of each length smaller than n.

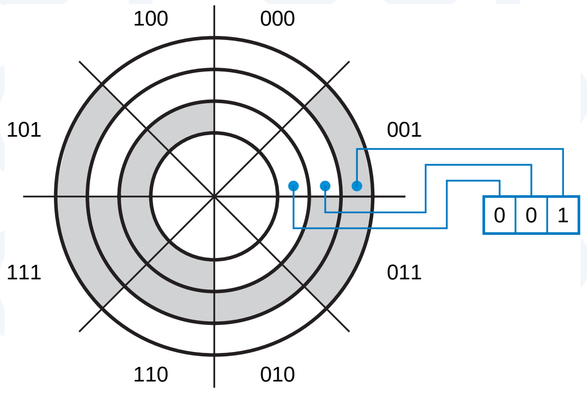


Figure 2-6
A mechanical encoding disk using a 3-bit Gray code.

The second method allows us to derive an n-bit Gray-code code word directly from the corresponding n-bit binary code word:

1. The bits of an n-bit binary or Gray-code code word are numbered from right to left, from 0 to $n - 1$.
2. Bit i of a Gray-code code word is 0 if bits i and $i + 1$ of the corresponding binary code word are the same, else bit i is 1. (When $i + 1 = n$, bit n of the binary code word is considered to be 0.)

Again, inspection of Table 2-10 shows that this is true for the 3-bit Gray code.

*2.12 Character Codes

As we showed in the preceding section, a string of bits need not represent a number, and in fact most of the information processed by computers is nonnumeric. The most common type of nonnumeric data is *text*, strings of characters from some character set. Each character is represented in the computer by a bit string according to an established convention.

text

The most commonly used character code is *ASCII* (pronounced *ASS key*), the American Standard Code for Information Interchange. ASCII represents each character with a 7-bit string, yielding a total of 128 different characters shown in Table 2-11. The code contains the uppercase and lowercase alphabet, numerals, punctuation, and various nonprinting control characters. Thus, the text string "Yeccch!" is represented by a rather innocuous-looking list of seven 7-bit numbers:

ASCII

1011001 1100101 1100011 1100011 1100011 1101000 0100001

2.13 Codes for Actions, Conditions, and States

The codes that we've described so far are generally used to represent things that we would probably consider to be "data"—things like numbers, positions, and characters. Programmers know that dozens of different data types can be used in a single computer program.

In digital system design, we often encounter nondata applications where a string of bits must be used to control an action, to flag a condition, or to represent the current state of the hardware. Probably the most commonly used type of code for such an application is a simple binary code. If there are n different actions, conditions, or states, we can represent them with a b-bit binary code with $b = \lceil \log_2 n \rceil$ bits. (The brackets $\lceil\ \rceil$ denote the *ceiling function*—the smallest integer greater than or equal to the bracketed quantity. Thus, b is the smallest integer such that $2^b \geq n$.)

$\lceil\ \rceil$
ceiling function

For example, consider a simple traffic-light controller. The signals at the intersection of a north-south (N-S) and an east-west (E-W) street might be in any

Table 2-11 American Standard Code for Information Interchange (ASCII), Standard No. X3.4-1968 of the American National Standards Institute.

		$b_6b_5b_4$ (column)							
$b_3b_2b_1b_0$	Row (hex)	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000	0	NUL	DLE	SP	0	@	P	‘	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	’	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Control codes

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronize
BEL	Bell	ETB	End transmitted block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete or rubout

Table 2-12 States in a traffic-light controller.

	Lights						
State	N-S green	N-S yellow	N-S red	E-W green	E-W yellow	E-W red	Code word
N-S go	ON	off	off	off	off	ON	000
N-S wait	off	ON	off	off	off	ON	001
N-S delay	off	off	ON	off	off	ON	010
E-W go	off	off	ON	ON	off	off	100
E-W wait	off	off	ON	off	ON	off	101
E-W delay	off	off	ON	off	off	ON	110

of the six states listed in Table 2-12. These states can be encoded in three bits, as shown in the last column of the table. Only six of the eight possible 3-bit code words are used, and the assignment of the six chosen code words to states is arbitrary, so many other encodings are possible. An experienced digital designer chooses a particular encoding to minimize circuit cost or to optimize some other parameter (like design time—there's no need to try billions and billions of possible encodings).

Another application of a binary code is illustrated in Figure 2-7(a). Here, we have a system with *n* devices, each of which can perform a certain action. The characteristics of the devices are such that they may be enabled to operate only one at a time. The control unit produces a binary-coded "device select" word with $\lceil \log_2 n \rceil$ bits to indicate which device is enabled at any time. The "device select" code word is applied to each device, which compares it with its own "device ID" to determine whether it is enabled.Although its code words have the minimum number of bits, a binary code isn't always the best choice for encoding actions, conditions, or states. Figure 2-7(b) shows how to control *n* devices with a *1-out-of-n code*, an *n*-bit code in which valid code words have one bit equal to 1 and the rest of the bits equal to 0. Each bit of the 1-out-of-*n* code word is connected directly to the enable input of a corresponding device. This simplifies the design of the devices, since they no longer have device IDs; they need only a single "enable" input bit.

1-out-of-n code

The code words of a 1-out-of-10 code were listed in Table 2-9. Sometimes an all-0s word may also be included in a 1-out-of-*n* code, to indicate that no device is selected. Another common code is an *inverted 1-out-of-n code*, in which valid code words have one 0~bit and the rest of the bits equal to 1.

inverted 1-out-of-n code

In complex systems, a combination of coding techniques may be used. For example, consider a system similar to Figure 2-7(b), in which each of the *n* devices contains up to *s* subdevices. The control unit could produce a device

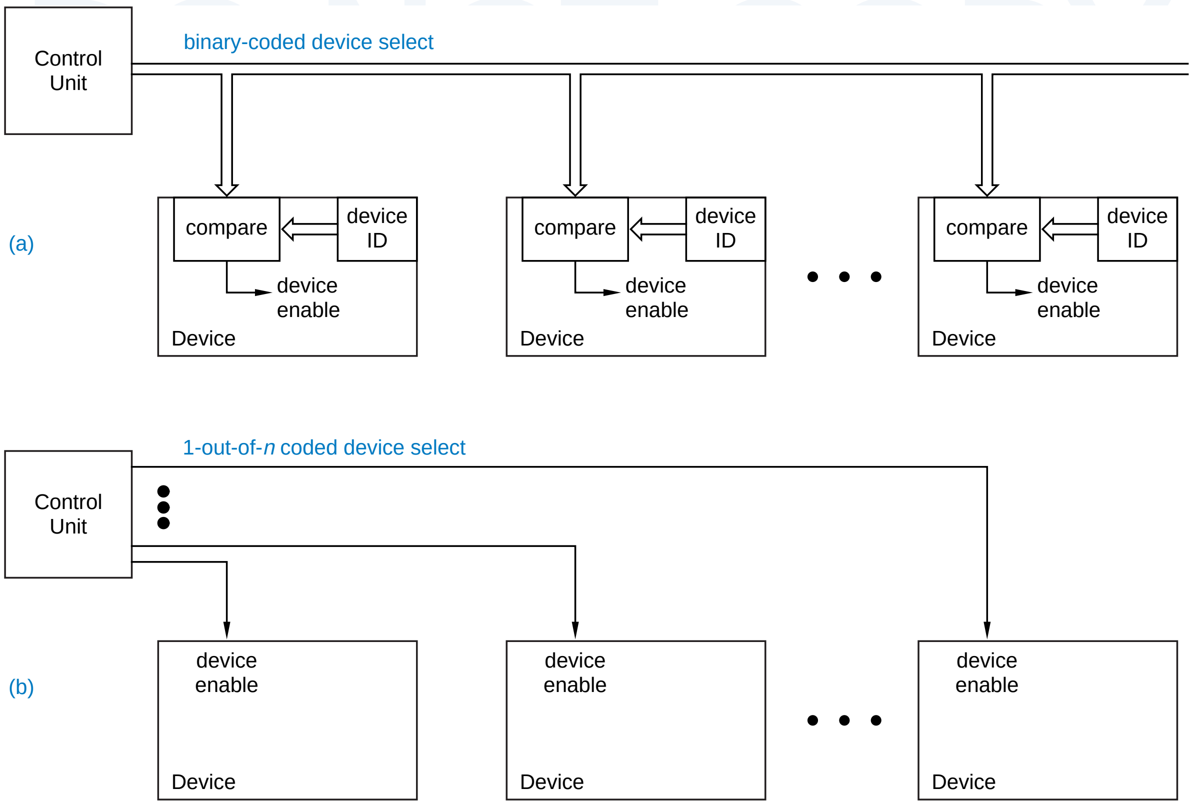


Figure 2-7 Control structure for a digital system with n devices: (a) using a binary code; (b) using a 1-out-of-n code.

select code word with a 1-out-of-n coded field to select a device, and a $\lceil \log_2 s \rceil$-bit binary-coded field to select one of the s subdevices of the selected device.

m-out-of-n code

An *m-out-of-n code* is a generalization of the 1-out-of-n code in which valid code words have m bits equal to 1 and the rest of the bits equal to 0. A valid m-out-of-n code word can be detected with an m-input AND gate, which produces a 1 output if all of its inputs are 1. This is fairly simple and inexpensive to do, yet for most values of m, an m-out-of-n code typically has far more valid code words than a 1-out-of-n code. The total number of code words is given by the binomial coefficient $\binom{n}{m}$, which has the value $\frac{n!}{m! \cdot (n-m)!}$. Thus, a 2-out-of-4 code has 6 valid code words, and a 3-out-of-10 code has 120.

8B10B code

An important variation of an m-out-of-n code is the *8B10B code* used in the 802.3z Gigabit Ethernet standard. This code uses 10 bits to represent 256 valid code words, or 8 bits worth of data. Most code words use a 5-out-of-10 coding. However, since $\binom{5}{10}$ is only 252, some 4- and 6-out-of-10 words are also used to complete the code in a very interesting way; more on this in Section 2.16.2.

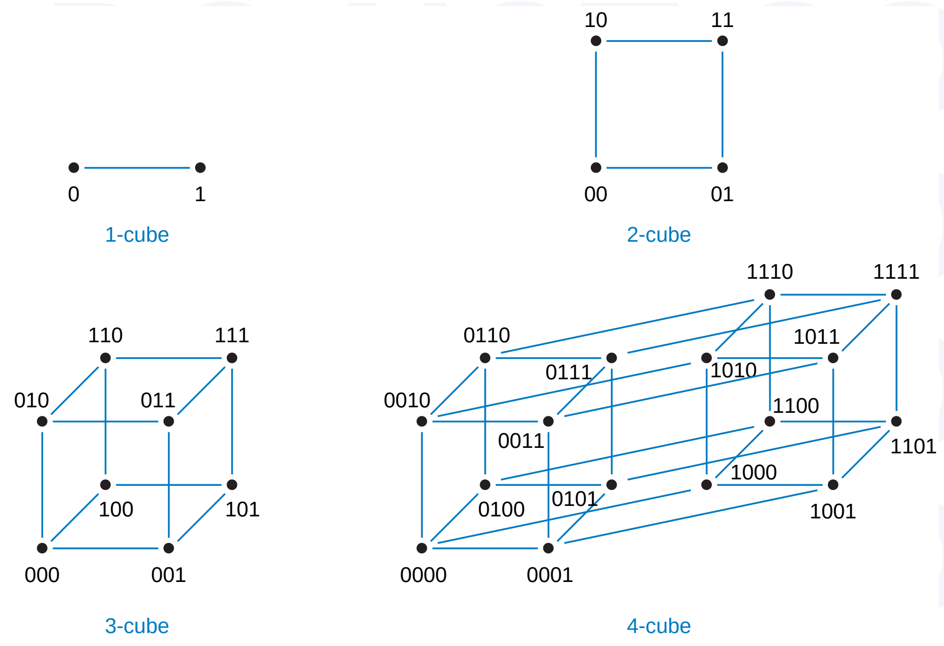


Figure 2-8
n-cubes for $n = 1, 2,$ $3,$ and 4.

*2.14 *n*-Cubes and Distance

n-cube

An *n*-bit string can be visualized geometrically, as a vertex of an object called an *n-cube*. Figure 2-8 shows *n*-cubes for $n = 1, 2, 3, 4$. An *n*-cube has 2^n vertices, each of which is labeled with an *n*-bit string. Edges are drawn so that each vertex is adjacent to *n* other vertices whose labels differ from the given vertex in only one bit. Beyond $n = 4$, *n*-cubes are really tough to draw.

For reasonable values of *n*, *n*-cubes make it easy to visualize certain coding and logic minimization problems. For example, the problem of designing an *n*-bit Gray code is equivalent to finding a path along the edges of an *n*-cube, a path that visits each vertex exactly once. The paths for 3- and 4-bit Gray codes are shown in Figure 2-9.

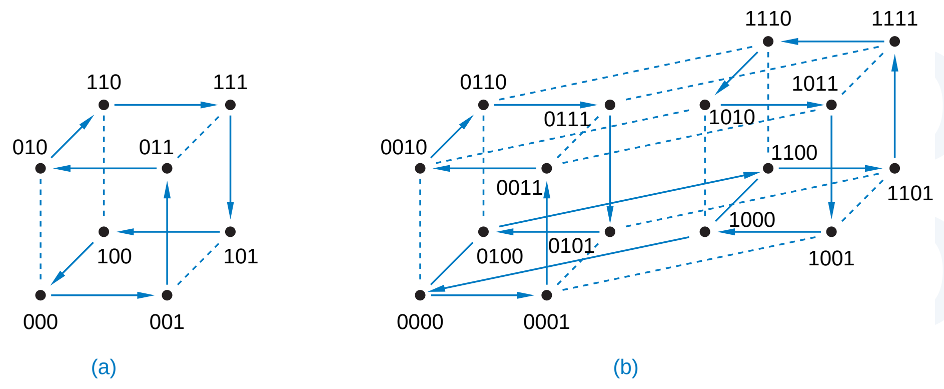


Figure 2-9
Traversing *n*-cubes in Gray-code order:
(a) 3-cube;
(b) 4-cube.

distance
Hamming distance

Cubes also provide a geometrical interpretation for the concept of *distance*, also called *Hamming distance*. The distance between two *n*-bit strings is the number of bit positions in which they differ. In terms of an *n*-cube, the distance is the minimum length of a path between the two corresponding vertices. Two adjacent vertices have distance 1; vertices 001 and 100 in the 3-cube have distance 2. The concept of distance is crucial in the design and understanding of error-detecting codes, discussed in the next section.

m-subcube

An *m-subcube* of an *n*-cube is a set of 2^m vertices in which $n - m$ of the bits have the same value at each vertex, and the remaining *m* bits take on all 2^m combinations. For example, the vertices (000, 010, 100, 110) form a 2-subcube of the 3-cube. This subcube can also be denoted by a single string, xx0, where "x" denotes that a particular bit is a *don't-care*; any vertex whose bits match in the non-x positions belongs to this subcube. The concept of subcubes is particularly useful in visualizing algorithms that minimize the cost of combinational logic functions, as we'll show in Section 4.4.

don't-care

*2.15 Codes for Detecting and Correcting Errors

error
failure
temporary failure
permanent failure

An *error* in a digital system is the corruption of data from its correct value to some other value. An error is caused by a physical *failure*. Failures can be either temporary or permanent. For example, a cosmic ray or alpha particle can cause a temporary failure of a memory circuit, changing the value of a bit stored in it. Letting a circuit get too hot or zapping it with static electricity can cause a permanent failure, so that it never works correctly again.

error model
independent error model
single error
multiple error

The effects of failures on data are predicted by *error models*. The simplest error model, which we consider here, is called the *independent error model*. In this model, a single physical failure is assumed to affect only a single bit of data; the corrupted data is said to contain a *single error*. Multiple failures may cause *multiple errors*—two or more bits in error—but multiple errors are normally assumed to be less likely than single errors.

2.15.1 Error-Detecting Codes

error-detecting code
noncode word

Recall from our definitions in Section 2.10 that a code that uses *n*-bit strings need not contain 2^n valid code words; this is certainly the case for the codes that we now consider. An *error-detecting code* has the property that corrupting or garbling a code word will likely produce a bit string that is not a code word (a *noncode word*).

A system that uses an error-detecting code generates, transmits, and stores only code words. Thus, errors in a bit string can be detected by a simple rule—if the bit string is a code word, it is assumed to be correct; if it is a noncode word, it contains an error.

An *n*-bit code and its error-detecting properties under the independent error model are easily explained in terms of an *n*-cube. A code is simply a subset

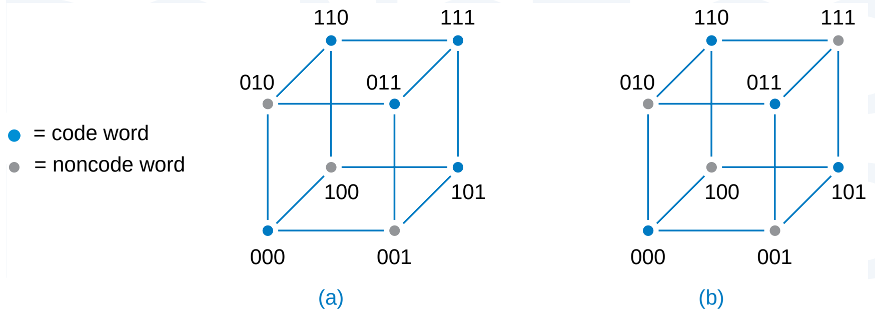


Figure 2-10 Code words in two different 3-bit codes: (a) minimum distance = 1, does not detect all single errors; (b) minimum distance = 2, detects all single errors.

of the vertices of the *n*-cube. In order for the code to detect all single errors, no code-word vertex can be immediately adjacent to another code-word vertex.

For example, Figure 2-10(a) shows a 3-bit code with five code words. Code word 111 is immediately adjacent to code words 110, 011 and 101. Since a single failure could change 111 to 110, 011 or 101 this code does not detect all single errors. If we make 111 a noncode word, we obtain a code that does have the single-error-detecting property, as shown in (b). No single error can change one code word into another.

The ability of a code to detect single errors can be stated in terms of the concept of distance introduced in the preceding section:

minimum distance

- A code detects all single errors if the *minimum distance* between all possible pairs of code words is 2.

information bit

parity bit

even-parity code

In general, we need *n* + 1 bits to construct a single-error-detecting code with 2*n* code words. The first *n* bits of a code word, called *information bits*, may be any of the 2*n* *n*-bit strings. To obtain a minimum-distance-2 code, we add one more bit, called a *parity bit*, that is set to 0 if there are an even number of 1s among the information bits, and to 1 otherwise. This is illustrated in the first two columns of Table 2-13 for a code with three information bits. A valid *n*+1-bit code word has an even number of 1s, and this code is called an *even-parity code*.

Table 2-13 Distance-2 codes with three information bits.

Information Bits	***Even-parity Code***	***Odd-parity Code***
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

odd-parity code
1-bit parity code

We can also construct a code in which the total number of 1s in a valid n+1-bit code word is odd; this is called an *odd-parity code* and is shown in the third column of the table. These codes are also sometimes called *1-bit parity codes*, since they each use a single parity bit.

The 1-bit parity codes do not detect 2-bit errors, since changing two bits does not affect the parity. However, the codes can detect errors in any *odd* number of bits. For example, if three bits in a code word are changed, then the resulting word has the wrong parity and is a noncode word. This doesn't help us much, though. Under the independent error model, 3-bit errors are much less likely than 2-bit errors, which are not detectable. Thus, practically speaking, the 1-bit parity codes' error detection capability stops after 1-bit errors. Other codes, with minimum distance greater than 2, can be used to detect multiple errors.

2.15.2 Error-Correcting and Multiple-Error-Detecting Codes

check bits

By using more than one parity bit, or *check bits*, according to some well-chosen rules, we can create a code whose minimum distance is greater than 2. Before showing how this can be done, let's look at how such a code can be used to correct single errors or detect multiple errors.

Suppose that a code has a minimum distance of 3. Figure 2-11 shows a fragment of the n-cube for such a code. As shown, there are at least two noncode words between each pair of code words. Now suppose we transmit code words

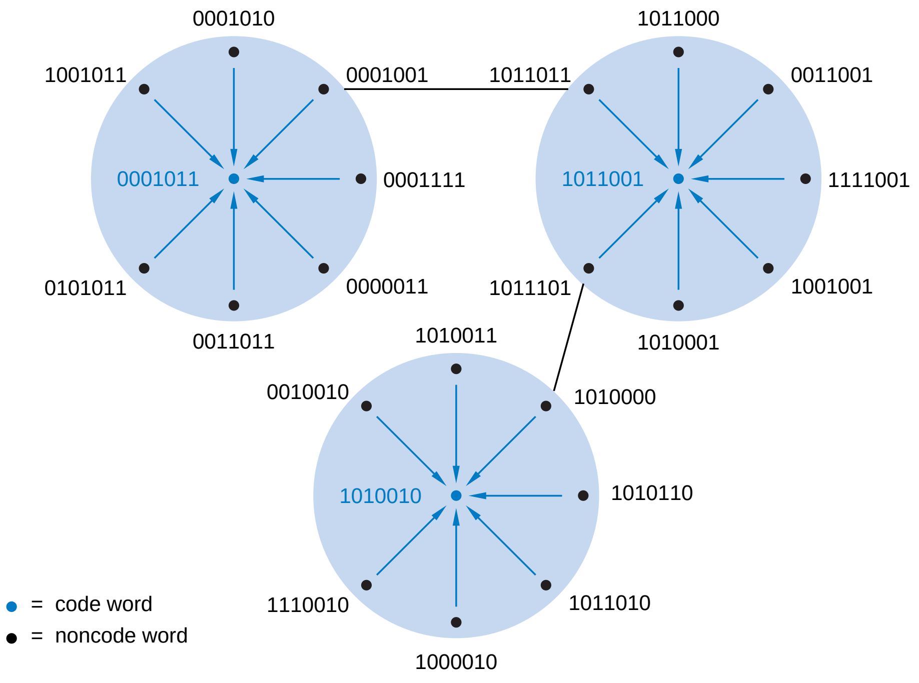


Figure 2-11 Some code words and noncode words in a 7-bit, distance-3 code.

and assume that failures affect at most one bit of each received code word. Then a received noncode word with a 1-bit error will be closer to the originally transmitted code word than to any other code word. Therefore, when we receive a noncode word, we can *correct* the error by changing the received noncode word to the nearest code word, as indicated by the arrows in the figure. Deciding which code word was originally transmitted to produce a received word is called *decoding*, and the hardware that does this is an error-correcting *decoder*.

error correction

decoding

decoder

A code that is used to correct errors is called an *error-correcting code*. In general, if a code has minimum distance $2c + 1$, it can be used to correct errors that affect up to c bits ($c = 1$ in the preceding example). If a code's minimum distance is $2c + d + 1$, it can be used to correct errors in up to c bits and to detect errors in up to d additional bits.

error-correcting code

For example, Figure 2-12(a) shows a fragment of the n-cube for a code with minimum distance 4 ($c = 1$, $d = 1$). Single-bit errors that produce noncode words 00101010 and 11010011 can be corrected. However, an error that produces 10100011 cannot be corrected, because no single-bit error can produce this noncode word, and either of two 2-bit errors could have produced it. So the code can detect a 2-bit error, but it cannot correct it.

When a noncode word is received, we don't know which code word was originally transmitted; we only know which code word is closest to what we've received. Thus, as shown in Figure 2-12(b), a 3-bit error may be "corrected" to the wrong value. The possibility of making this kind of mistake may be acceptable if 3-bit errors are very unlikely to occur. On the other hand, if we are concerned about 3-bit errors, we can change the decoding policy for the code. Instead of trying to correct errors, we just flag all noncode words as uncorrectable errors. Thus, as shown in (c), we can use the same distance-4 code to detect up to 3-bit errors but correct no errors ($c = 0$, $d = 3$).

2.15.3 Hamming Codes

In 1950, R. W. Hamming described a general method for constructing codes with a minimum distance of 3, now called *Hamming codes*. For any value of i, his method yields a 2^i–1-bit code with i check bits and $2^i - 1 - i$ information bits. Distance-3 codes with a smaller number of information bits are obtained by deleting information bits from a Hamming code with a larger number of bits.

Hamming code

The bit positions in a Hamming code word can be numbered from 1 through 2^i–1. In this case, any position whose number is a power of 2 is a check bit, and the remaining positions are information bits. Each check bit is grouped with a subset of the information bits, as specified by a *parity-check matrix*. As

parity-check matrix

DECISIONS, DECISIONS The names *decoding* and *decoder* make sense, since they are just distance-1 perturbations of *deciding* and *decider*.

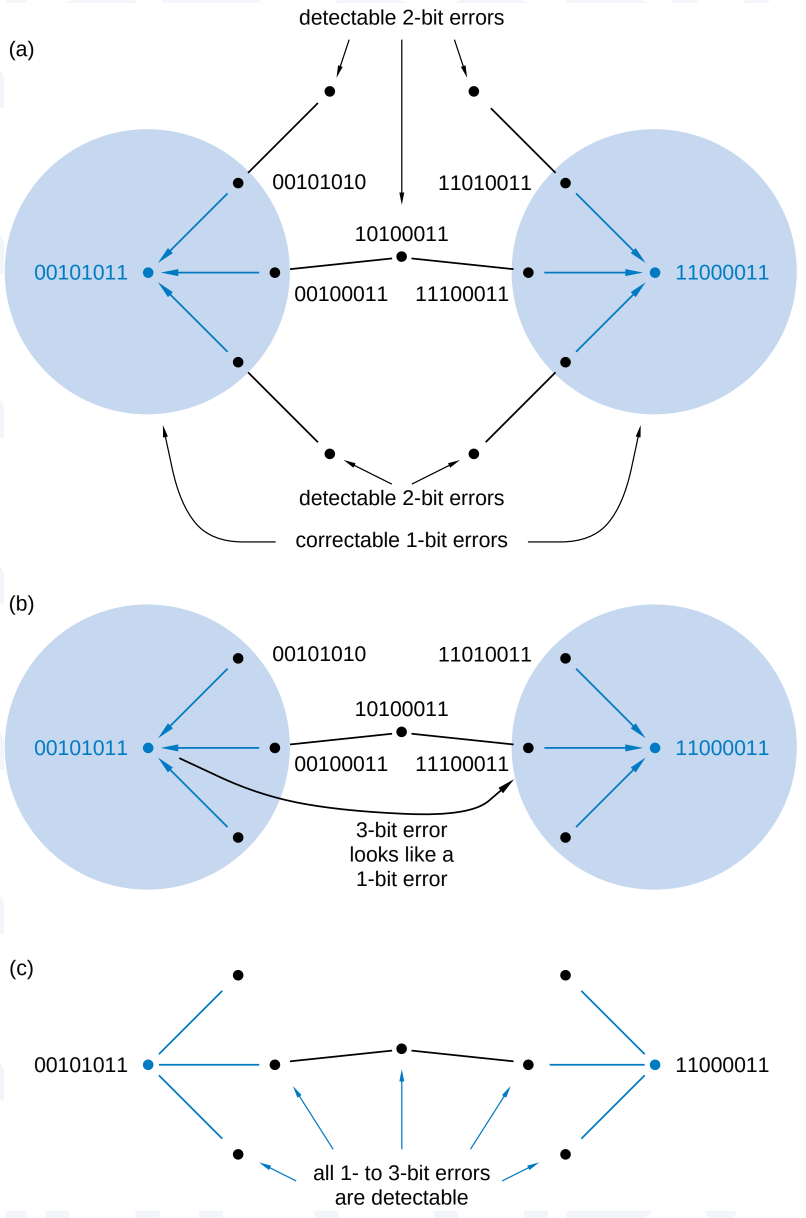


Figure 2-12
Some code words and noncode words in an 8-bit, distance-4 code:
(a) correcting 1-bit and detecting 2-bit errors;
(b) incorrectly "correcting" a 3-bit error;
(c) correcting no errors but detecting up to 3-bit errors.

shown in Figure 2-13(a), each check bit is grouped with the information positions whose numbers have a 1 in the same bit when expressed in binary. For example, check bit 2 (010) is grouped with information bits 3 (011), 6 (110), and 7 (111). For a given combination of information-bit values, each check bit is chosen to produce even parity, that is, so the total number of 1s in its group is even.

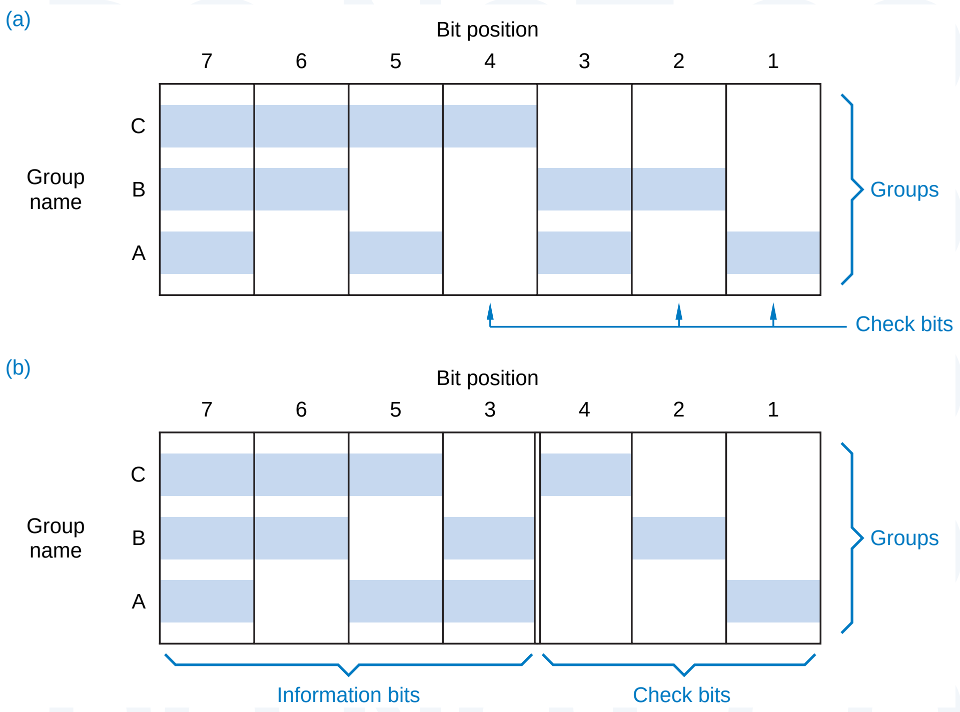


Figure 2-13 Parity-check matrices for 7-bit Hamming codes: (a) with bit positions in numerical order; (b) with check bits and information bits separated.

Traditionally, the bit positions of a parity-check matrix and the resulting code words are rearranged so that all of the check bits are on the right, as in Figure 2-13(b). The first two columns of Table 2-14 list the resulting code words.

We can prove that the minimum distance of a Hamming code is 3 by proving that at least a 3-bit change must be made to a code word to obtain another code word. That is, we'll prove that a 1-bit or 2-bit change in a code word yields a noncode word.

If we change one bit of a code word, in position *j*, then we change the parity of every group that contains position *j*. Since every information bit is contained in at least one group, at least one group has incorrect parity, and the result is a noncode word.

What happens if we change two bits, in positions *j* and *k*? Parity groups that contain both positions *j* and *k* will still have correct parity, since parity is unaffected when an even number of bits are changed. However, since *j* and *k* are different, their binary representations differ in at least one bit, corresponding to one of the parity groups. This group has only one bit changed, resulting in incorrect parity and a noncode word.

If you understand this proof, you should also see how the position numbering rules for constructing a Hamming code are a simple consequence of the proof. For the first part of the proof (1-bit errors), we required that the position numbers be nonzero. And for the second part (2-bit errors), we required that no

Table 2-14 Code words in distance-3 and distance-4 Hamming codes with four information bits.

Minimum-distance-3 code		Minimum-distance-4 code	
Information Bits	**Parity Bits**	**Information Bits**	**Parity Bits**
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

two positions have the same number. Thus, with an *i*-bit position number, you can construct a Hamming code with up to $2^i - 1$ bit positions.

error-correcting decoder

The proof also suggests how we can design an *error-correcting decoder* for a received Hamming code word. First, we check all of the parity groups; if all have even parity, then the received word is assumed to be correct. If one or more groups have odd parity, then a single error is assumed to have occurred. The pattern of groups that have odd parity (called the *syndrome*) must match one of the columns in the parity-check matrix; the corresponding bit position is assumed to contain the wrong value and is complemented. For example, using the code defined by Figure 2-13(b), suppose we receive the word 0101011. Groups B and C have odd parity, corresponding to position 6 of the parity-check matrix (the

syndrome

syndrome is 110, or 6). By complementing the bit in position 6 of the received word, we determine that the correct word is 0001011.

A distance-3 Hamming code can easily be modified to increase its minimum distance to 4. We simply add one more check bit, chosen so that the parity of all the bits, including the new one, is even. As in the 1-bit even-parity code, this bit ensures that all errors affecting an odd number of bits are detectable. In particular, any 3-bit error is detectable. We already showed that 1- and 2-bit errors are detected by the other parity bits, so the minimum distance of the modified code must be 4.

Distance-3 and distance-4 Hamming codes are commonly used to detect and correct errors in computer memory systems, especially in large mainframe computers where memory circuits account for the bulk of the system's failures. These codes are especially attractive for very wide memory words, since the required number of parity bits grows slowly with the width of the memory word, as shown in Table 2-15.

Table 2-15 Word sizes of distance-3 and distance-4 Hamming codes.

	Minimum-distance-3 Codes		*Minimum-distance-4 Codes*	
Information Bits	***Parity Bits***	***Total Bits***	***Parity Bits***	***Total Bits***
1	2	3	3	4
≤ 4	3	≤ 7	4	≤ 8
≤ 11	4	≤ 15	5	≤ 16
≤ 26	5	≤ 31	6	≤ 32
≤ 57	6	≤ 63	7	≤ 64
≤ 120	7	≤ 127	8	≤ 128

2.15.4 CRC Codes

Beyond Hamming codes, many other error-detecting and -correcting codes have been developed. The most important codes, which happen to include Hamming codes, are the *cyclic redundancy check (CRC) codes*. A rich set of knowledge has been developed for these codes, focused both on their error detecting and correcting properties and on the design of inexpensive encoders and decoders for them (see References).

cyclic redundancy check (CRC) code

Two important applications of CRC codes are in disk drives and in data networks. In a disk drive, each block of data (typically 512 bytes) is protected by a CRC code, so that errors within a block can be detected and, in some drives, corrected. In a data network, each packet of data ends with check bits in a CRC code. The CRC codes for both applications were selected because of their burst-error detecting properties. In addition to single-bit errors, they can detect multi-bit errors that are clustered together within the disk block or packet. Such errors are more likely than errors of randomly distributed bits, because of the likely physical causes of errors in the two applications—surface defects in disc drives and noise bursts in communication links.

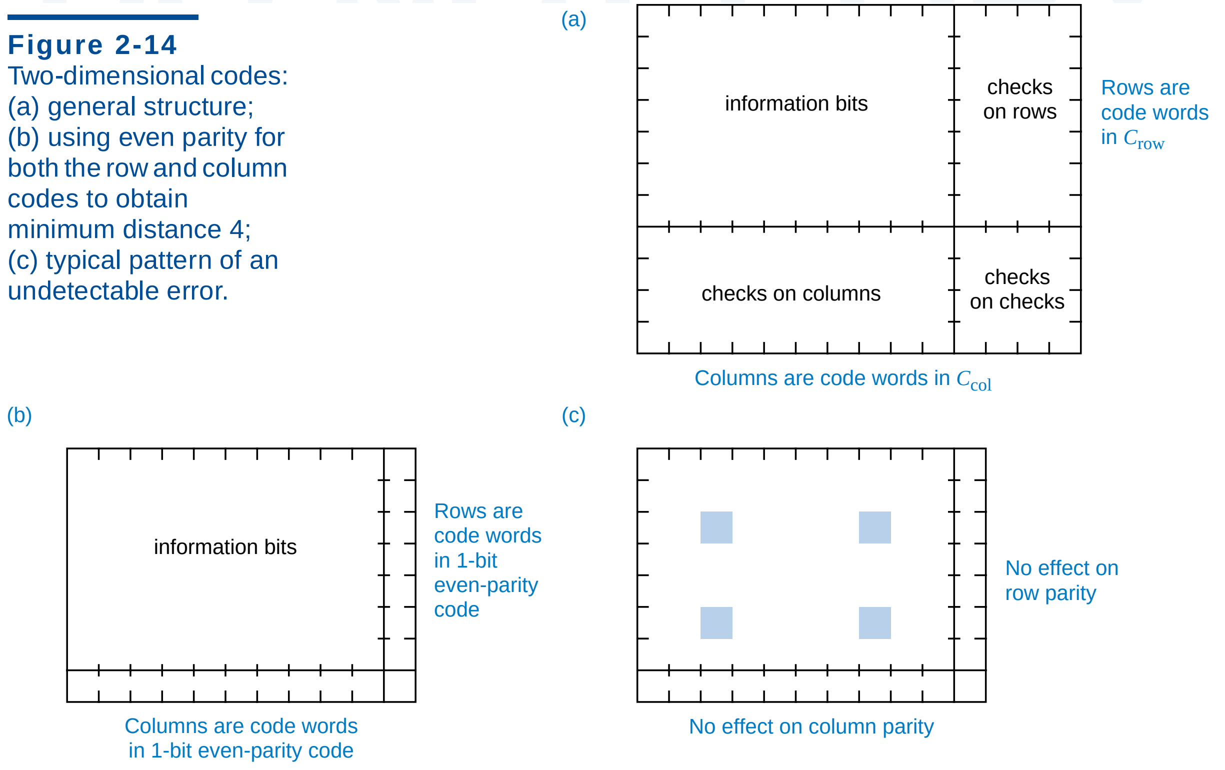
2.15.5 Two-Dimensional Codes

two-dimensional code

Another way to obtain a code with large minimum distance is to construct a *two-dimensional code*, as illustrated in Figure 2-14(a). The information bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns. A code C_{row} with minimum distance d_{row} is used for the rows, and a possibly different code C_{col} with minimum distance d_{col} is used for the columns. That is, the row-parity bits are selected so that each row is a code word in C_{row} and the column-parity bits are selected so that each column is a code word in C_{col}. (The "corner" parity bits can be chosen according to either code.) The minimum distance of the two-dimensional code is the product of d_{row} and d_{col}; in fact, two-dimensional codes are sometimes called *product codes*.

product code

Figure 2-14 Two-dimensional codes: (a) general structure; (b) using even parity for both the row and column codes to obtain minimum distance 4; (c) typical pattern of an undetectable error.


(a)
information bits
checks on rows
Rows are code words in C_{row}
checks on columns
checks on checks
Columns are code words in C_{col}
(b)
information bits
Rows are code words in 1-bit even-parity code
Columns are code words in 1-bit even-parity code
(c)
No effect on row parity
No effect on column parity

As shown in Figure 2-14(b), the simplest two-dimensional code uses 1-bit even-parity codes for the rows and columns, and has a minimum distance of $2 \cdot 2$, or 4. You can easily prove that the minimum distance is 4 by convincing yourself that any pattern of one, two, or three bits in error causes incorrect parity of a row or a column or both. In order to obtain an undetectable error, at least four bits must be changed in a rectangular pattern as in (c).

The error detecting and correcting procedures for this code are straightforward. Assume we are reading information one row at a time. As we read each row, we check its row code. If an error is detected in a row, we can't tell which bit is wrong from the row check alone. However, assuming only one row is bad, we can reconstruct it by forming the bit-by-bit Exclusive OR of the columns, omitting the bad row, but including the column-check row.

To obtain an even larger minimum distance, a distance-3 or -4 Hamming code can be used for the row or column code or both. It is also possible to construct a code in three or more dimensions, with minimum distance equal to the product of the minimum distances in each dimension.

An important application of two-dimensional codes is in RAID storage systems. *RAID* stands for "redundant array of inexpensive disks." In this scheme, n+1 identical disk drives are used to store n disks worth of data. For example, eight 8-Gigabyte drives could be use to store 64 Gigabytes of nonredundant data, and a ninth 8-gigabyte drive would be used to store checking information.

RAID

Figure 2-15 shows the general scheme of a two-dimensional code for a RAID system; each disk drive is considered to be a row in the code. Each drive stores m blocks of data, where a block typically contains 512 bytes. For example, an 8-gigabyte drive would store about 16 million blocks. As shown in the figure, each block includes its own check bits in a CRC code, to detect errors within that block. The first n drives store the nonredundant data. Each block in drive n+1

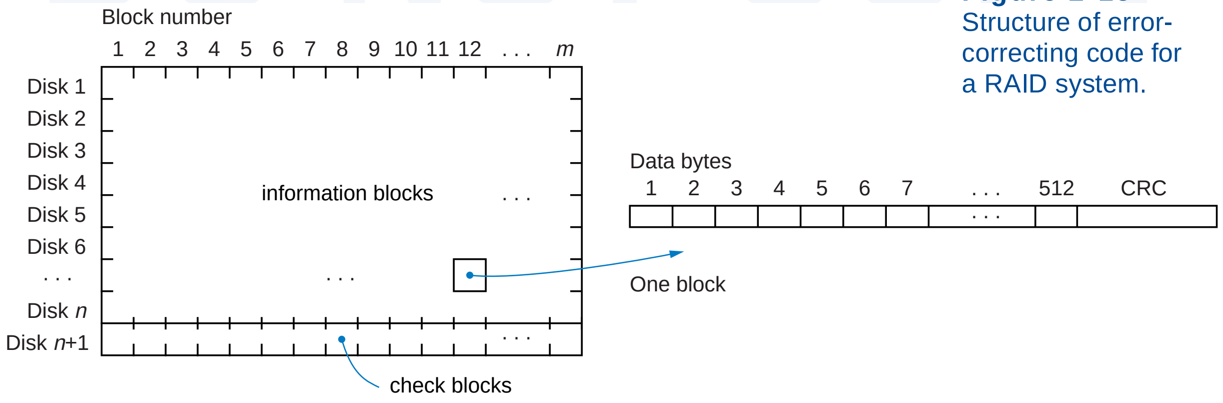


Figure 2-15
Structure of error-correcting code for a RAID system.

stores parity bits for the corresponding blocks in the first *n* drives. That is, each bit *i* in drive *n*+1 block *b* is chosen so that there are an even number of 1s in block *b* bit position *i* across all the drives.

In operation, errors in the information blocks are detected by the CRC code. Whenever an error is detected in a block on one of the drives, the correct contents of that block can be constructed simply by computing the parity of the corresponding blocks in all the other drives, including drive *n*+1. Although this requires *n* extra disk read operations, it's better than losing your data! Write operations require extra disk accesses as well, to update the corresponding check block when an information block is written (see Exercise 2.46). Since disk writes are much less frequent than reads in typical applications, this overhead usually is not a problem.

2.15.6 Checksum Codes

The parity-checking operation that we've used in the previous subsections is essentially modulo-2 addition of bits—the sum modulo 2 of a group of bits is 0 if the number of 1s in the group is even, and 1 if it is odd. The technique of modular addition can be extended to other bases besides 2 to form check digits.

For example, a computer stores information as a set of 8-bit bytes. Each byte may be considered to have a decimal value from 0 to 255. Therefore, we can use modulo-256 addition to check the bytes. We form a single check byte, called a *checksum*, that is the sum modulo 256 of all the information bytes. The resulting *checksum code* can detect any single *byte* error, since such an error will cause a recomputed sum of bytes to disagree with the checksum.

checksum
checksum code

Checksum codes can also use a different modulus of addition. In particular, checksum codes using modulo-255, ones'-complement addition are important because of their special computational and error detecting properties, and because they are used to check packet headers in the ubiquitous Internet Protocol (IP) (see References).

ones'-complement checksum code

2.15.7 *m*-out-of-*n* Codes

The 1-out-of-*n* and *m*-out-of-*n* codes that we introduced in Section 2.13 have a minimum distance of 2, since changing only one bit changes the total number of 1s in a code word and therefore produces a noncode word.

These codes have another useful error-detecting property—they detect unidirectional multiple errors. In a *unidirectional error*, all of the erroneous bits change in the same direction (0s change to 1s, or vice versa). This property is very useful in systems where the predominant error mechanism tends to change all bits in the same direction.

unidirectional error

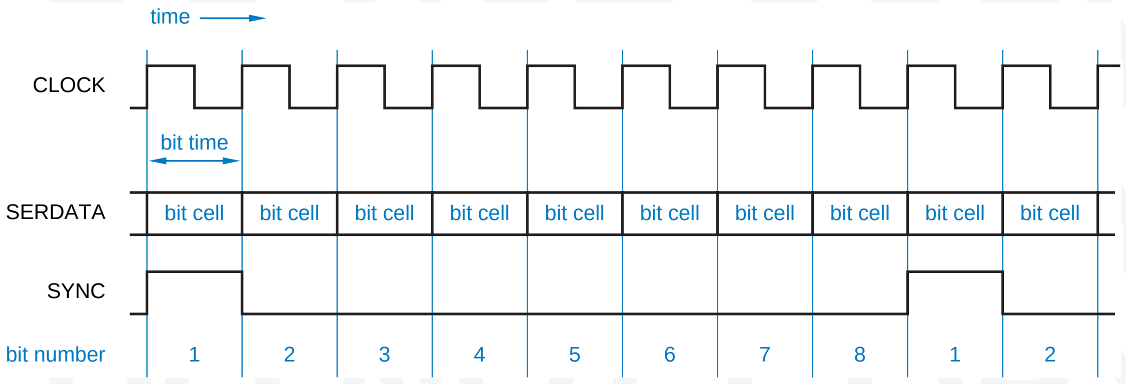


Figure 2-16 Basic concepts for serial data transmission.

2.16 Codes for Serial Data Transmission and Storage

2.16.1 Parallel and Serial Data

Most computers and other digital systems transmit and store data in a *parallel* format. In parallel data transmission, a separate signal line is provided for each bit of a data word. In parallel data storage, all of the bits of a data word can be written or read simultaneously.

parallel data

Parallel formats are not cost-effective for some applications. For example, parallel transmission of data bytes over the telephone network would require eight phone lines, and parallel storage of data bytes on a magnetic disk would require a disk drive with eight separate read/write heads. *Serial* formats allow data to be transmitted or stored one bit at a time, reducing system cost in many applications.

serial data

Figure 2-16 illustrates some of the basic ideas in serial data transmission. A repetitive clock signal, named CLOCK in the figure, defines the rate at which bits are transmitted, one bit per clock cycle. Thus, the *bit rate* in bits per second (bps) numerically equals the clock frequency in cycles per second (hertz, or Hz).

bit rate, bps

The reciprocal of the bit rate is called the *bit time* and numerically equals the clock period in seconds (s). This amount of time is reserved on the serial data line (named SERDATA in the figure) for each bit that is transmitted. The time occupied by each bit is sometimes called a *bit cell*. The format of the actual signal that appears on the line during each bit cell depends on the *line code*. In the simplest line code, called *Non-Return-to-Zero (NRZ)*, a 1 is transmitted by placing a 1 on the line for the entire bit cell, and a 0 is transmitted as a 0. However, more complex line codes have other rules, as discussed in the next subsection.

bit time

bit cell

line code

Non-Return-to-Zero (NRZ)

Regardless of the line code, a serial data transmission or storage system needs some way of identifying the significance of each bit in the serial stream. For example, suppose that 8-bit bytes are transmitted serially. How can we tell which is the first bit of each byte? A *synchronization signal*, named SYNC in Figure 2-16, provides the necessary information; it is 1 for the first bit of each byte.

synchronization signal

Evidently, we need a minimum of three signals to recover a serial data stream: a clock to define the bit cells, a synchronization signal to define the word boundaries, and the serial data itself. In some applications, like the interconnection of modules in a computer or telecommunications system, a separate wire is used for each of these signals, since reducing the number of wires per connection from *n* to three is savings enough. We'll give an example of a 3-wire serial data system in Section 8.5.4.

In many applications, the cost of having three separate signals is still too high (e.g., three phone lines, three read/write heads). Such systems typically combine all three signals into a single serial data stream and use sophisticated analog and digital circuits to recover the clock and synchronization information from the data stream.

*2.16.2 Serial Line Codes

The most commonly used line codes for serial data are illustrated in Figure 2-17. In the NRZ code, each bit value is sent on the line for the entire bit cell. This is the simplest and most reliable coding scheme for short distance transmission. However, it generally requires a clock signal to be sent along with the data to define the bit cells. Otherwise, it is not possible for the receiver to determine how many 0s or 1s are represented by a continuous 0 or 1 level. For example, without a clock to define the bit cells, the NRZ waveform in Figure 2-17 might be erroneously interpreted as 01010.

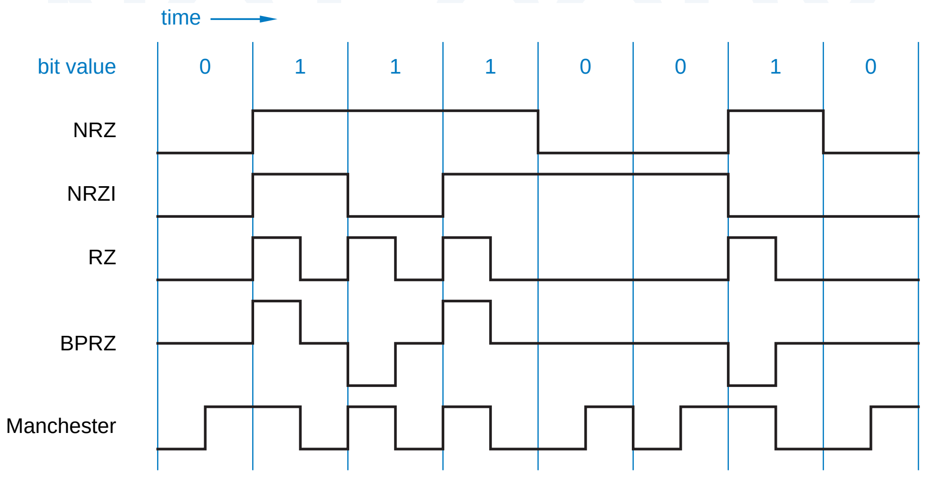


Figure 2-17
Commonly used line codes for serial data.

digital phase-locked loop (DPLL)

A *digital phase-locked loop (DPLL)* is an analog/digital circuit that can be used to recover a clock signal from a serial data stream. The DPLL works only if the serial data stream contains enough 0-to-1 and 1-to-0 transitions to give the DPLL "hints" about when the original clock transitions took place. With NRZ-coded data, the DPLL works only if the data does not contain any long, continuous streams of 1s or 0s.

transition-sensitive media

Some serial transmission and storage media are *transition sensitive*; they cannot transmit or store absolute 0 or 1 levels, only transitions between two discrete levels. For example, a magnetic disk or tape stores information by changing the polarity of the medium's magnetization in regions corresponding to the stored bits. When the information is recovered, it is not feasible to determine the absolute magnetization polarity of a region, only that the polarity changes between one region and the next.

Non-Return-to-Zero Invert-on-1s (NRZI)

Data stored in NRZ format on transition-sensitive media cannot be recovered unambiguously; the data in Figure 2-17 might be interpreted as 01110010 or 10001101. The *Non-Return-to-Zero Invert-on-1s (NRZI)* code overcomes this limitation by sending a 1 as the opposite of the level that was sent during the previous bit cell, and a 0 as the same level. A DPLL can recover the clock from NRZI-coded data as long as the data does not contain any long, continuous streams of 0s.

Return-to-Zero (RZ)

The Return-to-Zero (RZ) code is similar to NRZ except that, for a 1 bit, the 1 level is transmitted only for a fraction of the bit time, usually 1/2. With this code, data patterns that contain a lot of 1s create lots of transitions for a DPLL to use to recover the clock. However, as in the other line codes, a string of 0s has no transitions, and a long string of 0s makes clock recovery impossible.

DC balance

Another requirement of some transmission media, such as high-speed fiber-optic links, is that the serial data stream be *DC balanced*. That is, it must have an equal number of 1s and 0s; any long-term DC component in the stream (created by have a lot more 1s than 0s or vice versa) creates a bias at the receiver that reduces its ability to distinguish reliably between 1s and 0s.

balanced code

Ordinarily, NRZ, NRZI or RZ data has no guarantee of DC balance; there's nothing to prevent a user data stream from having a long string of words with more than 1s than 0s or vice versa. However, DC balance can still be achieved using a few extra bits to code the user data in a *balanced code*, in which each code word has an equal number of 1s and 0s, and then sending these code words in NRZ format.

For example, in Section 2.13 we introduced the 8B10B code, which codes 8 bits of user data into 10 bits in a mostly 5-out-of-10 code. Recall that there are only 252 5-out-of-10 code words, but there are another $\binom{4}{10} = 210$ 4-out-of-10 code words and an equal number of 6-out-of-10 code words. Of course, these code words aren't quite DC balanced. The 8B10B code solves this problem by associating with each 8-bit value to be encoded a *pair* of unbalanced code words, one 4-out-of-10 ("light") and the other 6-out-of-10 ("heavy"). The coder also

KILO-, MEGA-, GIGA-, TERA-

The prefixes K (kilo-), M (mega-), G (giga-), and T (tera-) mean 10^3, 10^6, 10^9, and 10^{12}, respectively, when referring to bps, hertz, ohms, watts, and most other engineering quantities. However, when referring to memory sizes, the prefixes mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}. Historically, the prefixes were co-opted for this purpose because memory sizes are normally powers of 2, and 2^{10} (1024) is very close to 1000,

Now, when somebody offers you 50 kilobucks a year for your first engineering job, it's up to you to negotiate what the prefix means!

running disparity

keeps track of the *running disparity*, a single bit of information indicating whether the last unbalanced code word that it transmitted was heavy or light. When it comes time to transmit another unbalanced code word, the coder selects the one of the pair with the opposite weight. This simple trick makes available 252 + 210 = 462 code words for the 8B10B to encode 8 bits of user data. Some of the "extra" code words are used to conveniently encode non-data conditions on the serial line, such as IDLE, SYNC, and ERROR. Not all the unbalanced code words are used. Also, some of the balanced code words, such as 0000011111, are not used either, in favor of unbalanced pairs that contain more transitions.

Bipolar Return-to-Zero (BPRZ)

Alternate Mark Inversion (AMI)

All of the preceding codes transmit or store only two signal levels. The *Bipolar Return-to-Zero (BPRZ)* code transmits three signal levels: +1, 0, and −1. The code is like RZ except that 1s are alternately transmitted as +1 and −1; for this reason, the code is also known as *Alternate Mark Inversion (AMI)*.

The big advantage of BPRZ over RZ is that it's DC balanced. This makes it possible to send BPRZ streams over transmission media that cannot tolerate a DC component, such as transformer-coupled phone lines. In fact, the BPRZ code has been used in T1 digital telephone links for decades, where analog speech signals are carried as streams of 8000 8-bit digital samples per second that are transmitted in BPRZ format on 64 Kbps serial channels.

zero-code suppression

As with RZ, it is possible to recover a clock signal from a BPRZ stream as long as there aren't too many 0s in a row. Although TPC (The Phone Company) has no control over what you say (at least, not yet), they still have a simple way of limiting runs of 0s. If one of the 8-bit bytes that results from sampling your analog speech pattern is all 0s, they simply change second-least significant bit to 1! This is called *zero-code suppression* and I'll bet you never noticed it. And this is also why, in many data applications of T1 links, you get only 56 Kbps of usable data per 64 Kbps channel; the LSB of each byte is always set to 1 to prevent zero-code suppression from changing the other bits.

Manchester

diphase

The last code in Figure 2-17 is called *Manchester* or *diphase* code. The major strength of this code is that, regardless of the transmitted data pattern, it provides at least one transition per bit cell, making it very easy to recover the clock. As shown in the figure, a 0 is encoded as a 0-to-1 transition in the middle

ABOUT TPC Watch the 1967 James Coburn movie, *The President's Analyst*, for an amusing view of TPC. With the growing pervasiveness of digital technology and cheap wireless communications, the concept of universal, *personal* connectivity to the phone network presented in the movie's conclusion has become much less far-fetched.

of the bit cell, and a 1 is encoded as a 1-to-0 transition. The Manchester code's major strength is also its major weakness. Since it has more transitions per bit cell than other codes, it also requires more media bandwidth to transmit a given bit rate. Bandwidth is not a problem in coaxial cable, however, which was used in the original Ethernet local area networks to carry Manchester-coded serial data at the rate of 10 Mbps (megabits per second).

References

The presentation in the first nine sections of this chapter is based on Chapter 4 of *Microcomputer Architecture and Programming*, by John F. Wakerly (Wiley, 1981). Precise, thorough, and entertaining discussions of these topics can also be found in Donald E. Knuth's *Seminumerical Algorithms*, 3rd edition (Addison-Wesley, 1997). Mathematically inclined readers will find Knuth's analysis of the properties of number systems and arithmetic to be excellent, and all readers should enjoy the insights and history sprinkled throughout the text.

Descriptions of digital logic circuits for arithmetic operations, as well as an introduction to properties of various number systems, appear in *Computer Arithmetic* by Kai Hwang (Wiley, 1979). *Decimal Computation* by Hermann Schmid (Wiley, 1974) contains a thorough description of techniques for BCD arithmetic.

An introduction to algorithms for binary multiplication and division and to floating-point arithmetic appears in *Microcomputer Architecture and Programming: The 68000 Family* by John F. Wakerly (Wiley, 1989). A more thorough discussion of arithmetic techniques and floating-point number systems can be found in *Introduction to Arithmetic for Digital Systems Designers* by Shlomo Waser and Michael J. Flynn (Holt, Rinehart and Winston, 1982).

finite fields

CRC codes are based on the theory of *finite fields*, which was developed by French mathematician Évariste Galois (1811–1832) shortly before he was killed in a duel with a political opponent. The classic book on error-detecting and error-correcting codes is *Error-Correcting Codes* by W. W. Peterson and E. J. Weldon, Jr. (MIT Press, 1972, 2nd ed.); however, this book is recommended only for mathematically sophisticated readers. A more accessible introduction can be found in *Error Control Coding: Fundamentals and Applications* by S. Lin and D. J. Costello, Jr. (Prentice Hall, 1983). Another recent, communication-oriented introduction to coding theory can be found in *Error-Control*

Techniques for Digital Communication by A. M. Michelson and A. H. Levesque (Wiley-Interscience, 1985). Hardware applications of codes in computer systems are discussed in *Error-Detecting Codes, Self-Checking Circuits, and Applications* by John F. Wakerly (Elsevier/North-Holland, 1978).

As shown in the above reference by Wakerly, ones'-complement checksum codes have the ability to detect long bursts of unidirectional errors; this is useful in communication channels where errors all tend to be in the same direction. The special computational properties of these codes also make them quite amenable to efficient checksum calculation by software programs, important for their use in the Internet Protocol; see RFC-1071 and RFC-1141.

An introduction to coding techniques for serial data transmission, including mathematical analysis of the performance and bandwidth requirements of several codes, appears in *Introduction to Communications Engineering* by R. M. Gagliardi (Wiley-Interscience, 1988, 2nd ed.). A nice introduction to the serial codes used in magnetic disks and tapes is given in *Computer Storage Systems and Technology* by Richard Matick (Wiley-Interscience, 1977).

The structure of the 8B10B code and the rationale behind it is explained nicely in the original IBM patent by Peter Franaszek and Albert Widmer, U.S. patent number 4,486,739 (1984). This and almost all U.S. patents issued after 1971 can be found on the web at `www.patents.ibm.com`. When you're done reading Franaszek, for a good time do a boolean search for inventor "`wakerly`".

Drill Problems

2.1 Perform the following number system conversions:

(a) $1101011_2 = ?_{16}$ (b) $174003_8 = ?_2$
(c) $10110111_2 = ?_{16}$ (d) $67.24_8 = ?_2$
(e) $10100.1101_2 = ?_{16}$ (f) $F3A5_{16} = ?_2$
(g) $11011001_2 = ?_8$ (h) $AB3D_{16} = ?_2$
(i) $101111.0111_2 = ?_8$ (j) $15C.38_{16} = ?_2$

2.2 Convert the following octal numbers into binary and hexadecimal:

(a) $1023_8 = ?_2 = ?_{16}$ (b) $761302_8 = ?_2 = ?_{16}$
(c) $163417_8 = ?_2 = ?_{16}$ (d) $552273_8 = ?_2 = ?_{16}$
(e) $5436.15_8 = ?_2 = ?_{16}$ (f) $13705.207_8 = ?_2 = ?_{16}$

2.3 Convert the following hexadecimal numbers into binary and octal:

(a) $1023_{16} = ?_2 = ?_8$ (b) $7E6A_{16} = ?_2 = ?_8$
(c) $ABCD_{16} = ?_2 = ?_8$ (d) $C350_{16} = ?_2 = ?_8$
(e) $9E36.7A_{16} = ?_2 = ?_8$ (f) $DEAD.BEEF_{16} = ?_2 = ?_8$

2.4 What are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

2.5 Convert the following numbers into decimal:

(a) $1101011_2 = ?_{10}$ (b) $174003_8 = ?_{10}$
(c) $10110111_2 = ?_{10}$ (d) $67.24_8 = ?_{10}$
(e) $10100.1101_2 = ?_{10}$ (f) $F3A5_{16} = ?_{10}$
(g) $12010_3 = ?_{10}$ (h) $AB3D_{16} = ?_{10}$
(i) $7156_8 = ?_{10}$ (j) $15C.38_{16} = ?_{10}$

2.6 Perform the following number system conversions:

(a) $125_{10} = ?_2$ (b) $3489_{10} = ?_8$
(c) $209_{10} = ?_2$ (d) $9714_{10} = ?_8$
(e) $132_{10} = ?_2$ (f) $23851_{10} = ?_{16}$
(g) $727_{10} = ?_5$ (h) $57190_{10} = ?_{16}$
(i) $1435_{10} = ?_8$ (j) $65113_{10} = ?_{16}$

2.7 Add the following pairs of binary numbers, showing all carries:

$$\begin{array}{lr@{\qquad}lr@{\qquad}lr@{\qquad}lr} \text{(a)} & 110101 & \text{(b)} & 101110 & \text{(c)} & 11011101 & \text{(d)} & 1110010 \\ & +\ \ 11001 & & +\ 100101 & & +\ \ 1100011 & & +\ 1101101 \\ \hline \end{array}$$

2.8 Repeat Drill 2.7 using subtraction instead of addition, and showing borrows instead of carries.

2.9 Add the following pairs of octal numbers:

$$\begin{array}{lr@{\qquad}lr@{\qquad}lr@{\qquad}lr} \text{(a)} & 1372 & \text{(b)} & 47135 & \text{(c)} & 175214 & \text{(d)} & 110321 \\ & +\ \ 4631 & & +\ \ 5125 & & +\ 152405 & & +\ 56573 \\ \hline \end{array}$$

2.10 Add the following pairs of hexadecimal numbers:

$$\begin{array}{lr@{\qquad}lr@{\qquad}lr@{\qquad}lr} \text{(a)} & 1372 & \text{(b)} & 4F1A5 & \text{(c)} & F35B & \text{(d)} & 1B90F \\ & +\ \ 4631 & & +\ \ B8D5 & & +\ \ 27E6 & & +\ \ C44E \\ \hline \end{array}$$

2.11 Write the 8-bit signed-magnitude, two’s-complement, and ones’-complement representations for each of these decimal numbers: +18, +115, +79, −49, −3, −100.

2.12 Indicate whether or not overflow occurs when adding the following 8-bit two’s-complement numbers:

$$\begin{array}{lr@{\qquad}lr@{\qquad}lr@{\qquad}lr} \text{(a)} & 11010100 & \text{(b)} & 10111001 & \text{(c)} & 01011101 & \text{(d)} & 00100110 \\ & +\ 10101011 & & +\ 11010110 & & +\ 00100001 & & +\ 01011010 \\ \hline \end{array}$$

2.13 How many errors can be detected by a code with minimum distance *d*?

2.14 What is the minimum number of parity bits required to obtain a distance-4, two-dimensional code with *n* information bits?

Exercises

2.15 Here's a problem to whet your appetite. What is the hexadecimal equivalent of 61453_{10}?

2.16 Each of the following arithmetic operations is correct in at least one number system. Determine possible radices of the numbers in each operation.

(a) 1234 + 5432 = 6666	(b) 41 / 3 = 13
(c) 33/3 = 11	(d) 23+44+14+32 = 223
(e) 302/20 = 12.1	(f) $\sqrt{41}$ = 5

2.17 The first expedition to Mars found only the ruins of a civilization. From the artifacts and pictures, the explorers deduced that the creatures who produced this civilization were four-legged beings with a tentacle that branched out at the end with a number of grasping "fingers." After much study, the explorers were able to translate Martian mathematics. They found the following equation:

$$5x^2 - 50x + 125 = 0$$

with the indicated solutions $x = 5$ and $x = 8$. The value $x = 5$ seemed legitimate enough, but $x = 8$ required some explanation. Then the explorers reflected on the way in which Earth's number system developed, and found evidence that the Martian system had a similar history. How many fingers would you say the Martians had? (From *The Bent of Tau Beta Pi*, February, 1956.)

2.18 Suppose a $4n$-bit number B is represented by an n-digit hexadecimal number H. Prove that the two's complement of B is represented by the 16's complement of H. Make and prove true a similar statement for octal representation.

2.19 Repeat Exercise 2.18 using the ones' complement of B and the 15s' complement of H.

2.20 Given an integer x in the range $-2n^{-1} \le x \le 2n^{-1} - 1$, we define $[x]$ to be the two's-complement representation of x, expressed as a positive number: $[x] = x$ if $x \ge 0$ and $[x] = 2n - |x|$ if $x < 0$, where $|x|$ is the absolute value of x. Let y be another integer in the same range as x. Prove that the two's-complement addition rules given in Section 2.6 are correct by proving that the following equation is always true:

$$[x + y] = ([x] + [y]) \text{ modulo } 2^n$$

(*Hints:* Consider four cases based on the signs of x and y. Without loss of generality, you may assume that $|x| \ge |y|$.)

2.21 Repeat Exercise 2.20 using appropriate expressions and rules for ones'-complement addition.

2.22 State an overflow rule for addition of two's-complement numbers in terms of counting operations in the modular representation of Figure 2-3.

2.23 Show that a two's-complement number can be converted to a representation with more bits by *sign extension*. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement representation of X, where $m > n$, can be obtained by appending $m - n$ copies of X's sign bit to the left of the n-bit representation of X.

2.24 Show that a two's-complement number can be converted to a representation with fewer bits by removing higher-order bits. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement number Y obtained by discarding the d leftmost bits of X represents the same number as X if and only if the discarded bits all equal the sign bit of Y.

2.25 Why is the punctuation of "two's complement" and "ones' complement" inconsistent? (See the first two citations in the References.)

2.26 A n-bit binary adder can be used to perform an n-bit unsigned subtraction operation $X - Y$, by performing the operation $X + Y + 1$, where X and Y are n-bit unsigned numbers and Y represents the bit-by-bit complement of Y. Demonstrate this fact as follows. First, prove that $(X - Y) = (X + Y + 1) - 2^n$. Second, prove that the carry out of the n-bit adder is the opposite of the borrow from the n-bit subtraction. That is, show that the operation $X - Y$ produces a borrow out of the MSB position if and only if the operation $X + Y + 1$ *does not* produce a carry out of the MSB position.

2.27 In most cases, the product of two n-bit two's-complement numbers requires fewer than $2n$ bits to represent it. In fact, there is only one case in which $2n$ bits are needed—find it.

2.28 Prove that a two's-complement number can be multiplied by 2 by shifting it one bit position to the left, with a carry of 0 into the least significant bit position and disregarding any carry out of the most significant bit position, assuming no overflow. State the rule for detecting overflow.

2.29 State and prove correct a technique similar to the one described in Exercise 2.28, for multiplying a ones'-complement number by 2.

2.30 Show how to subtract BCD numbers, by stating the rules for generating borrows and applying a correction factor. Show how your rules apply to each of the following subtractions: 9 – 3, 5 – 7, 4 – 9, 1 – 8.

2.31 How many different 3-bit binary state encodings are possible for the traffic-light controller of Table 2-12?

2.32 List all of the "bad" boundaries in the mechanical encoding disc of Figure 2-5, where an incorrect position may be sensed.

2.33 As a function of n, how many "bad" boundaries are there in a mechanical encoding disc that uses an n-bit binary code?

2.34 On-board altitude transponders on commercial and private aircraft use Gray code to encode the altitude readings that are transmitted to air traffic controllers. Why?

2.35 An incandescent light bulb is stressed every time it is turned on, so in some applications the lifetime of the bulb is limited by the number of on/off cycles rather than the total time it is illuminated. Use your knowledge of codes to suggest a way to double the lifetime of 3-way bulbs in such applications.

2.36 As a function of n, how many different distinct subcubes of an n-cube are there?

2.37 Find a way to draw a 3-cube on a sheet of paper (or other two-dimensional object) so that none of the lines cross, or prove that it's impossible.

2.38 Repeat Exercise 2.37 for a 4-cube.

2.39 Write a formula that gives the number of *m*-subcubes of an *n*-cube for a specific value of *m*. (Your answer should be a function of *n* and *m*.)

2.40 Define parity groups for a distance-3 Hamming code with 11 information bits.

2.41 Write the code words of a Hamming code with one information bit.

2.42 Exhibit the pattern for a 3-bit error that is not detected if the “corner” parity bits are not included in the two-dimensional codes of Figure 2-14.

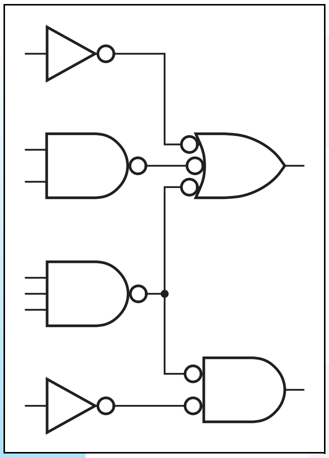
2.43 The *rate of a code* is the ratio of the number of information bits to the total number of bits in a code word. High rates, approaching 1, are desirable for efficient transmission of information. Construct a graph comparing the rates of distance-2 parity codes and distance-3 and -4 Hamming codes for up to 100 information bits.

2.44 Which type of distance-4 code has a higher rate—a two-dimensional code or a Hamming code? Support your answer with a table in the style of Table 2-15, including the rate as well as the number of parity and information bits of each code for up to 100 information bits.

2.45 Show how to construct a distance-6 code with four information bits. Write a list of its code words.

2.46 Describe the operations that must be performed in a RAID system to write new data into information block *b* in drive *d*, so the data can be recovered in the event of an error in block *b* in any drive. Minimize the number of disk accesses required.

2.47 In the style of Figure 2-17, draw the waveforms for the bit pattern 10101110 when sent serially using the NRZ, NRZI, RZ, BPRZ, and Manchester codes, assuming that the bits are transmitted in order from left to right.



chapter 4

Combinational Logic Design Principles

Logic circuits are classified into two types, "combinational" and "sequential." A *combinational logic circuit* is one whose outputs depend only on its current inputs. The rotary channel selector knob on an old-fashioned television is like a combinational circuit—its "output" selects a channel based only on the current position of the knob ("input").

The outputs of a *sequential logic circuit* depend not only on the current inputs, but also on the past sequence of inputs, possibly arbitrarily far back in time. The channel selector controlled by the up and down pushbuttons on a TV or VCR is a sequential circuit—the channel selection depends on the past sequence of up/down pushes, at least since when you started viewing 10 hours before, and perhaps as far back as when you first powered-up the device. Sequential circuits are discussed in Chapters xx through yy.

A combinational circuit may contain an arbitrary number of logic gates and inverters but no feedback loops. A *feedback loop* is a signal path of a circuit that allows the output of a gate to propagate back to the input of that same gate; such a loop generally creates sequential circuit behavior.

In combinational circuit *analysis* we start with a logic diagram, and proceed to a formal description of the function performed by that circuit, such as a truth table or a logic expression. In *synthesis*, we do the reverse, starting with a formal description and proceeding to a logic diagram.

SYNTHESIS VS. DESIGN

Logic circuit design is a superset of synthesis, since in a real design problem we usually start out with an informal (word or thought) description of the circuit. Often the most challenging and creative part of design is to formalize the circuit description, defining the circuit's input and output signals and specifying its functional behavior by means of truth tables and equations. Once we've created the formal circuit description, we can usually follow a "turn-the-crank" synthesis procedure to obtain a logic diagram for a circuit with the required functional behavior. The material in the first four sections of this chapter is the basis for "turn-the-crank" procedures, whether the crank is turned by hand or by a computer. The last two sections describe actual design languages, ABEL and VHDL. When we create a design using one of these languages, a computer program can perform the synthesis steps for us. In later chapters we'll encounter many examples of the real design process.

Combinational circuits may have one or more outputs. Most analysis and synthesis techniques can be extended in an obvious way from single-output to multiple-output circuits (e.g., "Repeat these steps for each output"). We'll also point out how some techniques can be extended in a not-so-obvious way for improved effectiveness in the multiple-output case.

The purpose of this chapter is to give you a solid theoretical foundation for the analysis and synthesis of combinational logic circuits, a foundation that will be doubly important later when we move on to sequential circuits. Although most of the analysis and synthesis procedures in this chapter are automated nowadays by computer-aided design tools, you need a basic understanding of the fundamentals to use the tools and to figure out what's wrong when you get unexpected or undesirable results.

With the fundamentals well in hand, it is appropriate next to understand how combinational functions can be expressed and analyzed using hardware description languages (HDLs). So, the last two sections of this chapter introduce basic features of ABEL and VHDL, which we'll use to design for all kinds of logic circuits throughout the balance of the text.

Before launching into a discussion of combinational logic circuits, we must introduce switching algebra, the fundamental mathematical tool for analyzing and synthesizing logic circuits of all types.

4.1 Switching Algebra

Boolean algebra

Formal analysis techniques for digital circuits have their roots in the work of an English mathematician, George Boole. In 1854, he invented a two-valued algebraic system, now called *Boolean algebra*, to "give expression . . . to the fundamental laws of reasoning in the symbolic language of a Calculus." Using this system, a philosopher, logician, or inhabitant of the planet Vulcan can formulate propositions that are true or false, combine them to make new propositions, and determine the truth or falsehood of the new propositions. For example, if we agree that "People who haven't studied this material are either failures or not nerds," and "No computer designer is a failure," then we can answer questions like "If you're a nerdy computer designer, then have you already studied this?"

Long after Boole, in 1938, Bell Labs researcher Claude E. Shannon showed how to adapt Boolean algebra to analyze and describe the behavior of circuits built from relays, the most commonly used digital logic elements of that time. In Shannon's *switching algebra*, the condition of a relay contact, open or closed, is represented by a variable X that can have one of two possible values, 0 or 1. In today's logic technologies, these values correspond to a wide variety of physical conditions—voltage HIGH or LOW, light off or on, capacitor discharged or charged, fuse blown or intact, and so on—as we detailed in Table 3-1 on page 77.

switching algebra

In the remainder of this section, we develop the switching algebra directly, using "first principles" and what we already know about the behavior of logic elements (gates and inverters). For more historical and/or mathematical treatments of this material, consult the References.

4.1.1 Axioms

In switching algebra we use a symbolic variable, such as X, to represent the condition of a logic signal. A logic signal is in one of two possible conditions—low or high, off or on, and so on, depending on the technology. We say that X has the value "0" for one of these conditions and "1" for the other.

For example, with the CMOS and TTL logic circuits in Chapter 3, the *positive-logic convention* dictates that we associate the value "0" with a LOW voltage and "1" with a HIGH voltage. The *negative-logic convention* makes the opposite association: 0 = HIGH and 1 = LOW. However, the choice of positive or negative logic has no effect on our ability to develop a consistent algebraic description of circuit behavior; it only affects details of the physical-to-algebraic abstraction, as we'll explain later in our discussion of "duality." For the moment, we may ignore the physical realities of logic circuits and pretend that they operate directly on the logic symbols 0 and 1.

positive-logic convention

negative-logic convention

The *axioms* (or *postulates*) of a mathematical system are a minimal set of basic definitions that we assume to be true, from which all other information about the system can be derived. The first two axioms of switching algebra embody the "digital abstraction" by formally stating that a variable X can take on only one of two values:

axiom

postulate

$$\text{(A1)}\quad X = 0 \quad \text{if } X \neq 1 \qquad \text{(A1}'\text{)}\quad X = 1 \quad \text{if } X \neq 0$$

Notice that we stated these axioms as a pair, with the only difference between A1 and A1′ being the interchange of the symbols 0 and 1. This is a characteristic of all the axioms of switching algebra, and is the basis of the “duality” principle that we’ll study later.

complement
prime (′)

In Section 3.3.3 we showed the design of an inverter, a logic circuit whose output signal level is the opposite (or *complement*) of its input signal level. We use a *prime (′)* to denote an inverter’s function. That is, if a variable X denotes an inverter’s input signal, then X′ denotes the value of a signal on the inverter’s output. This notation is formally specified in the second pair of axioms:

(A2) If $X = 0$, then $X' = 1$ (A2’) If $X = 1$, then $X' = 0$

algebraic operator
expression
NOT operation

As shown in Figure 4-1, the output of an inverter with input signal X may have an arbitrary signal name, say Y. However, algebraically, we write $Y = X'$ to say “signal Y always has the opposite value as signal X.” The prime (′) is an *algebraic operator*, and X′ is an *expression*, which you can read as “X prime” or “NOT X.” This usage is analogous to what you’ve learned in programming languages, where if J is an integer variable, then −J is an expression whose value is 0 − J. Although this may seem like a small point, you’ll learn that the distinction between signal names (X, Y), expressions (X′), and equations ($Y = X'$) is very important when we study documentation standards and software tools for logic design. In the logic diagrams in this book, we maintain this distinction by writing signal names in black and expressions in color.

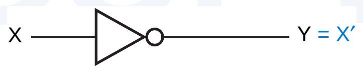


Figure 4-1
Signal naming and algebraic notation for an inverter.

logical multiplication
multiplication dot (·)

In Section 3.3.6 we showed how to build a 2-input CMOS AND gate, a circuit whose output is 1 if both of its inputs are 1. The function of a 2-input AND gate is sometimes called *logical multiplication* and is symbolized algebraically by a *multiplication dot (·)*. That is, an AND gate with inputs X and Y has an output signal whose value is $X \cdot Y$, as shown in Figure 4-2(a). Some authors, especially mathematicians and logicians, denote logical multiplication with a wedge $X \vee Y$). We follow standard engineering practice by using the dot ($X \cdot Y$). When

NOTE ON NOTATION

The notations X, ~X, and ¬X are also used by some authors to denote the complement of X. The overbar notation is probably the most widely used and the best looking typographically. However, we use the prime notation to get you used to writing logic expressions on a single text line without the more graphical overbar, and to force you to parenthesize complex complemented subexpressions—because this is what you’ll have to do when you use HDLs and other tools.

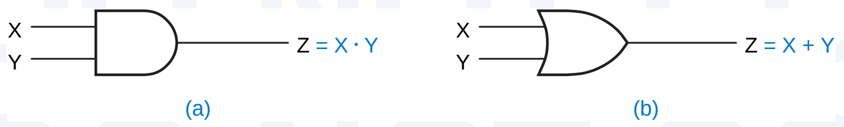


Figure 4-2
Signal naming and algebraic notation:
(a) AND gate;
(b) OR gate.

we study hardware design languages (HDLs), we'll encounter several other symbols that are used to denote the same thing.

We also described in Section 3.3.6 how to build a 2-input CMOS OR gate, a circuit whose output is 1 if either of its inputs is 1. The function of a 2-input OR gate is sometimes called *logical addition* and is symbolized algebraically by a plus sign (+). An OR gate with inputs X and Y has an output signal whose value is X + Y, as shown in Figure 4-2(b). Some authors denote logical addition with a vee (X ∧ Y), but we follow the standard engineering practice of using the plus sign (X + Y). Once again, other symbols may be used in HDLs. By convention, in a logic expression involving both multiplication and addition, multiplication has *precedence*, just as in integer expressions in conventional programming languages. That is, the expression W · X + Y · Z is equivalent to (W · X) + (Y · Z).

logical addition

precedence

The last three pairs of axioms state the formal definitions of the AND and OR operations by listing the output produced by each gate for each possible input combination:

AND operation
OR operation

(A3)	$0 \cdot 0 = 0$	(A3′)	$1 + 1 = 1$
(A4)	$1 \cdot 1 = 1$	(A4′)	$0 + 0 = 0$
(A5)	$0 \cdot 1 = 1 \cdot 0 = 0$	(A5′)	$1 + 0 = 0 + 1 = 1$

The five pairs of axioms, A1–A5 and A1′–A5′, completely define switching algebra. All other facts about the system can be proved using these axioms as a starting point.

JUXT A MINUTE…

Older texts use simple *juxtaposition* (XY) to denote logical multiplication, but we don't. In general, juxtaposition is a clear notation only when signal names are limited to a single character. Otherwise, is XY a logical product or a two-character signal name? One-character variable names are common in algebra, but in real digital design problems, we prefer to use multicharacter signal names that mean something. Thus, we need a separator between names, and the separator might just as well be a multiplication dot rather than a space. The HDL equivalent of the multiplication dot (often * or &) is absolutely required when logic formulas are written in a hardware design language.

Table 4-1 Switching-algebra theorems with one variable.

(T1)	$X + 0 = X$	(T1′)	$X \cdot 1 = X$	(Identities)
(T2)	$X + 1 = 1$	(T2′)	$X \cdot 0 = 0$	(Null elements)
(T3)	$X + X = X$	(T3′)	$X \cdot X = X$	(Idempotency)
(T4)	$(X')' = X$			(Involution)
(T5)	$X + X' = 1$	(T5′)	$X \cdot X' = 0$	(Complements)

4.1.2 Single-Variable Theorems

During the analysis or synthesis of logic circuits, we often write algebraic expressions that characterize a circuit's actual or desired behavior. Switching-algebra *theorems* are statements, known to be always true, that allow us to manipulate algebraic expressions to allow simpler analysis or more efficient synthesis of the corresponding circuits. For example, the theorem $X + 0 = X$ allows us to substitute every occurrence of $X + 0$ in an expression with X.

theorem

Table 4-1 lists switching-algebra theorems involving a single variable X. How do we know that these theorems are true? We can either prove them ourselves or take the word of someone who has. OK, we're in college now, let's learn how to prove them.

Most theorems in switching algebra are exceedingly simple to prove using a technique called *perfect induction*. Axiom A1 is the key to this technique—since a switching variable can take on only two different values, 0 and 1, we can prove a theorem involving a single variable X by proving that it is true for both $X = 0$ and $X = 1$. For example, to prove theorem T1, we make two substitutions:

perfect induction

$[X = 0]$	$0 + 0 = 0$	true, according to axiom A4′
$[X = 1]$	$1 + 0 = 1$	true, according to axiom A5′

All of the theorems in Table 4-1 can be proved using perfect induction, as you're asked to do in the Drills 4.2 and 4.3.

4.1.3 Two- and Three-Variable Theorems

Switching-algebra theorems with two or three variables are listed in Table 4-2. Each of these theorems is easily proved by perfect induction, by evaluating the theorem statement for the four possible combinations of X and Y, or the eight possible combinations of X, Y, and Z.

The first two theorem pairs concern commutativity and associativity of logical addition and multiplication and are identical to the commutative and associative laws for addition and multiplication of integers and reals. Taken together, they indicate that the parenthesization or order of terms in a logical sum or logical product is irrelevant. For example, from a strictly algebraic point of view, an expression such as $W \cdot X \cdot Y \cdot Z$ is ambiguous; it should be written as $(W \cdot (X \cdot (Y \cdot Z)))$ or $(((W \cdot X) \cdot Y) \cdot Z)$ or $(W \cdot X) \cdot (Y \cdot Z)$ (see Exercise 4.29). But the theorems tell us that the ambiguous form of the expression is OK

Table 4-2 Switching-algebra theorems with two or three variables.

(T6)	$X + Y = Y + X$	(T6′)	$X \cdot Y = Y \cdot X$	(Commutativity)
(T7)	$(X + Y) + Z = X + (Y + Z)$	(T7′)	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	(Associativity)
(T8)	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$	(T8′)	$(X + Y) \cdot (X + Z) = X + Y \cdot Z$	(Distributivity)
(T9)	$X + X \cdot Y = X$	(T9′)	$X \cdot (X + Y) = X$	(Covering)
(T10)	$X \cdot Y + X \cdot Y' = X$	(T10′)	$(X + Y) \cdot (X + Y') = X$	(Combining)
(T11)	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$			(Consensus)
(T11′)	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$			

because we get the same results in any case. We even could have rearranged the order of the variables (e.g., $X \cdot Z \cdot Y \cdot W$) and gotten the same results.

As trivial as this discussion may seem, it is very important because it forms the theoretical basis for using logic gates with more than two inputs. We defined $\cdot$ and $+$ as *binary operators*—operators that combine *two* variables. Yet we use 3-input, 4-input, and larger AND and OR gates in practice. The theorems tell us we can connect gate inputs in any order; in fact, many printed-circuit-board and ASIC layout programs take advantage of this. We can use either one *n*-input gate or $(n-1)$ 2-input gates interchangeably, though propagation delay and cost are likely to be higher with multiple 2-input gates.

binary operator

Theorem T8 is identical to the distributive law for integers and reals—that is, logical multiplication distributes over logical addition. Hence, we can "multiply out" an expression to obtain a sum-of-products form, as in the example below:

$$V \cdot (W + X) \cdot (Y + Z) = V \cdot W \cdot Y + V \cdot W \cdot Z + V \cdot X \cdot Y + V \cdot X \cdot Z$$

However, switching algebra also has the unfamiliar property that the reverse is true—logical addition distributes over logical multiplication—as demonstrated by theorem T8′. Thus, we can also "add out" an expression to obtain a product-of-sums form:

$$(V \cdot W \cdot X) + (Y \cdot Z) = (V + Y) \cdot (V + Z) \cdot (W + Y) \cdot (W + Z) \cdot (X + Y) \cdot (X + Z)$$

Theorems T9 and T10 are used extensively in the minimization of logic functions. For example, if the subexpression $X + X \cdot Y$ appears in a logic expression, the *covering theorem* T9 says that we need only include X in the expression; X is said to *cover* $X \cdot Y$. The *combining theorem* T10 says that if the subexpression $X \cdot Y + X \cdot Y'$ appears in an expression, we can replace it with X. Since Y must be 0 or 1, either way the original subexpression is 1 if and only if X is 1.

covering theorem
cover
combining theorem

Although we could easily prove T9 by perfect induction, the truth of T9 is more obvious if we prove it using the other theorems that we've proved so far:

$$\begin{aligned} X + X \cdot Y &= X \cdot 1 + X \cdot Y && \text{(according to T1}'\text{)} \\ &= X \cdot (1 + Y) && \text{(according to T8)} \\ &= X \cdot 1 && \text{(according to T2)} \\ &= X && \text{(according to T1}'\text{)} \end{aligned}$$

Likewise, the other theorems can be used to prove T10, where the key step is to use T8 to rewrite the left-hand side as $X \cdot (Y + Y')$.

consensus theorem
consensus

Theorem T11 is known as the *consensus theorem*. The $Y \cdot Z$ term is called the *consensus* of $X \cdot Y$ and $X' \cdot Z$. The idea is that if $Y \cdot Z$ is 1, then either $X \cdot Y$ or $X' \cdot Z$ must also be 1, since Y and Z are both 1 and either X or X' must be 1. Thus. the $Y \cdot Z$ term is redundant and may be dropped from the right-hand side of T11. The consensus theorem has two important applications. It can be used to eliminate certain timing hazards in combinational logic circuits, as we'll see in Section 4.5. And it also forms the basis of the iterative-consensus method of finding prime implicants (see References).

In all of the theorems, it is possible to replace each variable with an arbitrary logic expression. A simple replacement is to complement one or more variables:

$$(X + Y') + Z' \quad = \quad X + (Y' + Z') \qquad \text{(based on T7)}$$

But more complex expressions may be substituted as well:

$$(V' + X) \cdot (W \cdot (Y' + Z)) + (V' + X) \cdot (W \cdot (Y' + Z))' \;=\; V' + X \qquad \text{(based on T10)}$$

4.1.4 *n*-Variable Theorems

finite induction
basis step
induction step

Several important theorems, listed in Table 4-3, are true for an arbitrary number of variables, *n*. Most of these theorems can be proved using a two-step method called *finite induction*—first proving that the theorem is true for $n = 2$ (the *basis step*) and then proving that if the theorem is true for $n = i$, then it is also true for $n = i + 1$ (the *induction step*). For example, consider the generalized idempotency theorem T12. For $n = 2$, T12 is equivalent to T3 and is therefore true. If it is true for a logical sum of *i* X's, then it is also true for a sum of $i + 1$ X's, according to the following reasoning:

$$\begin{aligned} X + X + X + \cdots + X &= X + (X + X + \cdots + X) && (i + 1 \text{ X's on either side}) \\ &= X + (X) && (\text{if T12 is true for } n = i) \\ &= X && \text{(according to T3)} \end{aligned}$$

Thus, the theorem is true for all finite values of *n*.

DeMorgan's theorems

DeMorgan's theorems (T13 and T13′) are probably the most commonly used of all the theorems of switching algebra. Theorem T13 says that an *n*-input

Table 4-3 Switching-algebra theorems with *n* variables.

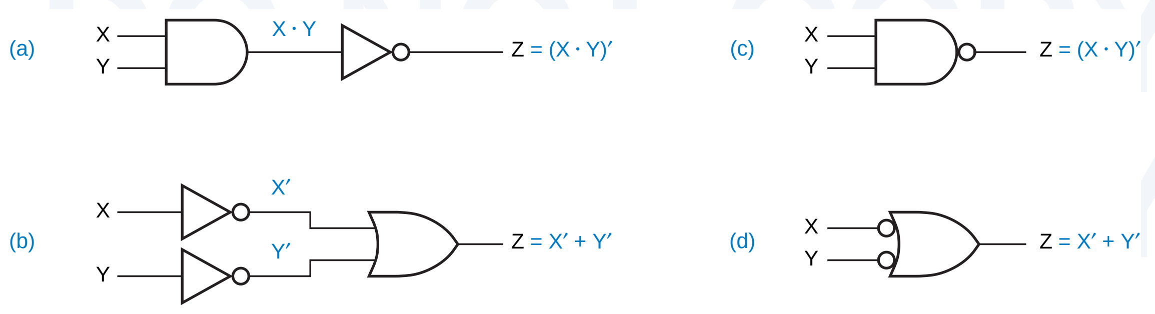
(T12)	$X + X + \cdots + X = X$	(Generalized idempotency)
(T12′)	$X \cdot X \cdot \cdots \cdot X = X$	
(T13)	$(X_1 \cdot X_2 \cdot \cdots \cdot X_n)' = X_1' + X_2' + \cdots + X_n'$	(DeMorgan’s theorems)
(T13′)	$(X_1 + X_2 + \cdots + X_n)' = X_1' \cdot X_2' \cdot \cdots \cdot X_n'$	
(T14)	$[F(X_1,X_2,\ldots,X_n,+,\cdot)]' = F(X_1',X_2',\ldots,X_n',\cdot,+)$	(Generalized DeMorgan’s theorem)
(T15)	$F(X_1,X_2,\ldots,X_n) = X_1 \cdot F(1,X_2,\ldots,X_n) + X_1' \cdot F(0,X_2,\ldots,X_n)$	(Shannon’s expansion theorems)
(T15′)	$F(X_1,X_2,\ldots,X_n) = [X_1 + F(0,X_2,\ldots,X_n)] \cdot [X_1' + F(1,X_2,\ldots,X_n)]$	

AND gate whose output is complemented is equivalent to an *n*-input OR gate whose inputs are complemented. That is, the circuits of Figure 4-3(a) and (b) are equivalent.

In Section 3.3.4 we showed how to build a CMOS NAND gate. The output of a NAND gate for any set of inputs is the complement of an AND gate’s output for the same inputs, so a NAND gate can have the logic symbol in Figure 4-3(c). However, the CMOS NAND circuit is not designed as an AND gate followed by a transistor inverter (NOT gate); it’s just a collection of transistors that happens to perform the AND-NOT function. In fact, theorem T13 tells us that the logic symbol in (d) denotes the same logic function (bubbles on the OR-gate inputs indicate logical inversion). That is, a NAND gate may be viewed as performing a NOT-OR function.

By observing the inputs and output of a NAND gate, it is impossible to determine whether it has been built internally as an AND gate followed by an inverter, as inverters followed by an OR gate, or as a direct CMOS realization, *because all NAND circuits perform precisely the same logic function*. Although the choice of symbol has no bearing on the functionality of a circuit, we’ll show in Section 5.1 that the proper choice can make the circuit’s function much easier to understand.

Figure 4-3 Equivalent circuits according to DeMorgan’s theorem T13: (a) AND-NOT; (b) NOT-OR; (c) logic symbol for a NAND gate; (d) equivalent symbol for a NAND gate.



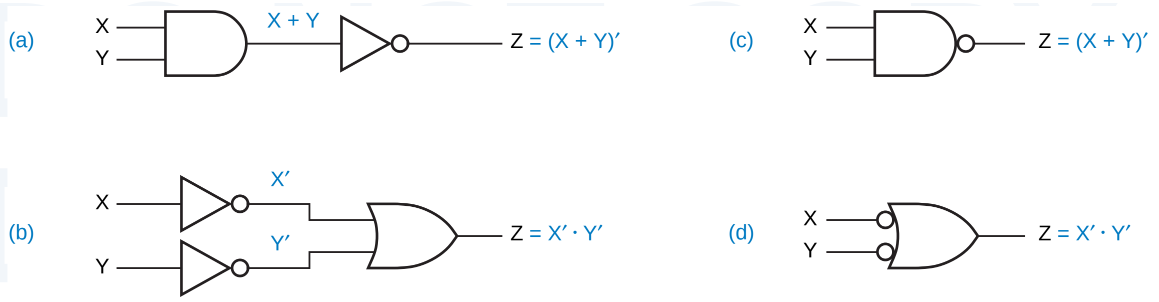


Figure 4-4 Equivalent circuits according to DeMorgan's theorem T13′: (a) OR-NOT; (b) NOT-AND; (c) logic symbol for a NOR gate; (d) equivalent symbol for a NOR gate.

A similar symbolic equivalence can be inferred from theorem T13′. As shown in Figure 4-4, a NOR gate may be realized as an OR gate followed by an inverter, or as inverters followed by an AND gate.

generalized DeMorgan's theorem

complement of a logic expression

Theorems T13 and T13′ are just special cases of a *generalized DeMorgan's theorem*, T14, that applies to an arbitrary logic expression F. By definition, the *complement of a logic expression*, denoted (F)′, is an expression whose value is the opposite of F's for every possible input combination. Theorem T14 is very important because it gives us a way to manipulate and simplify the complement of an expression.

Theorem T14 states that, given any *n*-variable logic expression, its complement can be obtained by swapping + and · and complementing all variables. For example, suppose that we have

$$\begin{aligned} F(W,X,Y,Z) &= (W' \cdot X) + (X \cdot Y) + (W \cdot (X' + Z')) \\ &= ((W)' \cdot X) + (X \cdot Y) + (W \cdot ((X)' + (Z)')) \end{aligned}$$

In the second line we have enclosed complemented variables in parentheses to remind you that the ′ is an operator, not part of the variable name. Applying theorem T14, we obtain

$$[F(W,X,Y,Z)]' = ((W')' + X') \cdot (X' + Y') \cdot (W' + ((X')' \cdot (Z')'))$$

Using theorem T4, this can be simplified to

$$[F(W,X,Y,Z)]' = (W) + X') \cdot (X' + Y') \cdot (W' + (X \cdot (Z))$$

In general, we can use theorem T14 to complement a parenthesized expression by swapping + and ·, complementing all uncomplemented variables, and uncomplementing all complemented ones.

The generalized DeMorgan's theorem T14 can be proved by showing that all logic functions can be written as either a sum or a product of subfunctions, and then applying T13 and T13′ recursively. However, a much more enlightening and satisfying proof can be based on the principle of duality, explained next.

4.1.5 Duality

We stated all of the axioms of switching algebra in pairs. The primed version of each axiom (e.g., A5′) is obtained from the unprimed version (e.g., A5) by simply swapping 0 and 1 and, if present, $\cdot$ and +. As a result, we can state the following *metatheorem*, a theorem about theorems:

metatheorem

Principle of Duality Any theorem or identity in switching algebra remains true if 0 and 1 are swapped and $\cdot$ and + are swapped throughout.

The metatheorem is true because the duals of all the axioms are true, so duals of all switching-algebra theorems can be proved using duals of the axioms.

After all, what's in a name, or in a symbol for that matter? If the software that was used to typeset this book had a bug, one that swapped $0 \leftrightarrow 1$ and $\cdot \leftrightarrow +$ throughout this chapter, you still would have learned exactly the same switching algebra; only the nomenclature would have been a little weird, using words like "product" to describe an operation that uses the symbol "+".

Duality is important because it doubles the usefulness of everything that you learn about switching algebra and manipulation of switching functions. Stated more practically, from a student's point of view, it halves the amount that you have to learn! For example, once you learn how to synthesize two-stage AND-OR logic circuits from sum-of-products expressions, you automatically know a dual technique to synthesize OR-AND circuits from product-of-sums expressions.

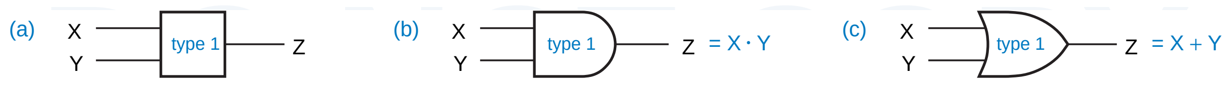
There is just one convention in switching algebra where we did not treat $\cdot$ and + identically, so duality does not necessarily hold true—can you figure out what it is before reading the answer below? Consider the following statement of theorem T9 and its clearly absurd "dual":

$$\begin{aligned} X + X \cdot Y &= X && \text{(theorem T9)} \\ X \cdot X + Y &= X && \text{(after applying the principle of duality)} \\ X + Y &= X && \text{(after applying theorem T3′)} \end{aligned}$$

Obviously the last line above is false—where did we go wrong? The problem is in operator precedence. We were able to write the left-hand side of the first line without parentheses because of our convention that $\cdot$ has precedence. However, once we applied the principle of duality, we should have given precedence to + instead, or written the second line as $X \cdot (X + Y) = X$. The best way to avoid problems like this is to parenthesize an expression fully before taking its dual.

Let us formally define the *dual of a logic expression*. If $F(X_1,X_2,\ldots,X_n,+,\cdot,')$ is a fully parenthesized logic expression involving the variables $X_1,X_2,\ldots,X_n$ and

dual of a logic expression



X	Y	Z
LOW	LOW	LOW
LOW	HIGH	LOW
HIGH	LOW	LOW
HIGH	HIGH	HIGH

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	Z
1	1	1
1	0	1
0	1	1
0	0	0

Figure 4-5 A “type-1”logic gate: (a) electrical function table; (b) logic function table and symbol with positive logic; (c) logic function table and symbol with negative logic.

the operators +, · , and ′, then the dual of F, written F^D, is the same expression with + and · swapped:

$$F^D(X_1,X_2,\ldots,X_n,+,\cdot,') = F(X_1,X_2,\ldots,X_n,\cdot,+,')$$

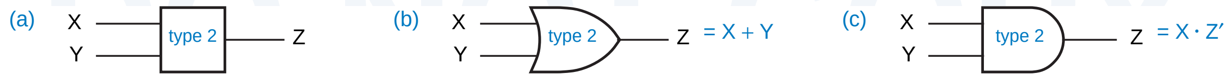
You already knew this, of course, but we wrote the definition in this way just to highlight the similarity between duality and the generalized DeMorgan’s theorem T14, which may now be restated as follows:

$$[F(X_1,X_2,\ldots,X_n)]' = F^D(X_1',X_2',\ldots,X_n')$$

Let’s examine this statement in terms of a physical network.

Figure 4-5(a) shows the electrical function table for a logic element that we’ll simply call a “type-1” gate. Under the positive-logic convention (LOW = 0 and HIGH = 1), this is an AND gate, but under the negative-logic convention (LOW = 1 and HIGH = 0), it is an OR gate, as shown in (b) and (c). We can also imagine a “type-2” gate, shown in Figure 4-6, that is a positive-logic OR or a negative-logic AND. Similar tables can be developed for gates with more than two inputs.

Figure 4-6 A “type-2” logic gate: (a) electrical function table; (b) logic function table and symbol with positive logic; (c) logic function table and symbol with negative logic.



X	Y	Z
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	HIGH

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	Z
1	1	1
1	0	0
0	1	0
0	0	0

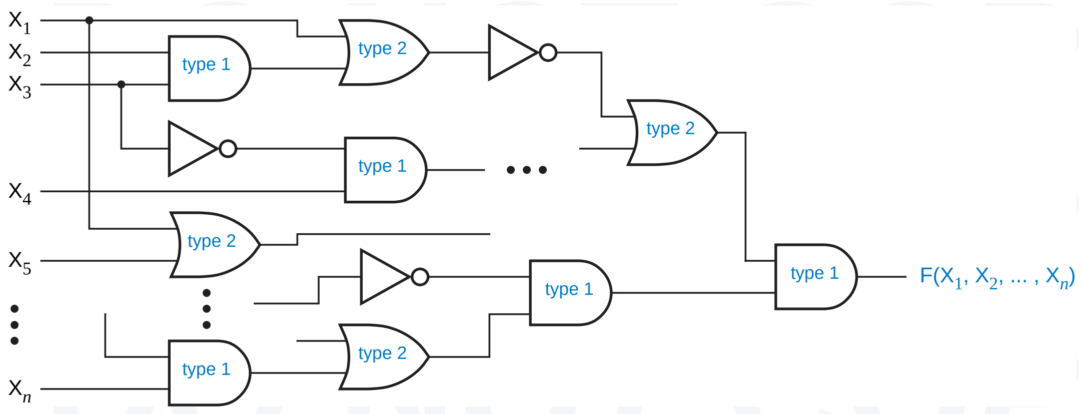


Figure 4-7 Circuit for a logic function using inverters and type-1 and type-2 gates under a positive-logic convention.

Suppose that we are given an arbitrary logic expression, $F(X_1,X_2,\ldots,X_n)$. Following the positive-logic convention, we can build a circuit corresponding to this expression using inverters for NOT operations, type-1 gates for AND, and type-2 gates for OR, as shown in Figure 4-7. Now suppose that, without changing this circuit, we simply change the logic convention from positive to negative. Then we should redraw the circuit as shown in Figure 4-8. Clearly, for every possible combination of input voltages (HIGH and LOW), the circuit still produces the same output voltage. However, from the point of view of switching algebra, the output value—0 or 1—is the opposite of what it was under the positive-logic convention. Likewise, each input value is the opposite of what it was. Therefore, for each possible input combination to the circuit in Figure 4-7, the output is the opposite of that produced by the opposite input combination applied to the circuit in Figure 4-8:

$$F(X_1,X_2,\ldots,X_n) \;=\; [F^D(X_1',X_2',\ldots,X_n')]'$$

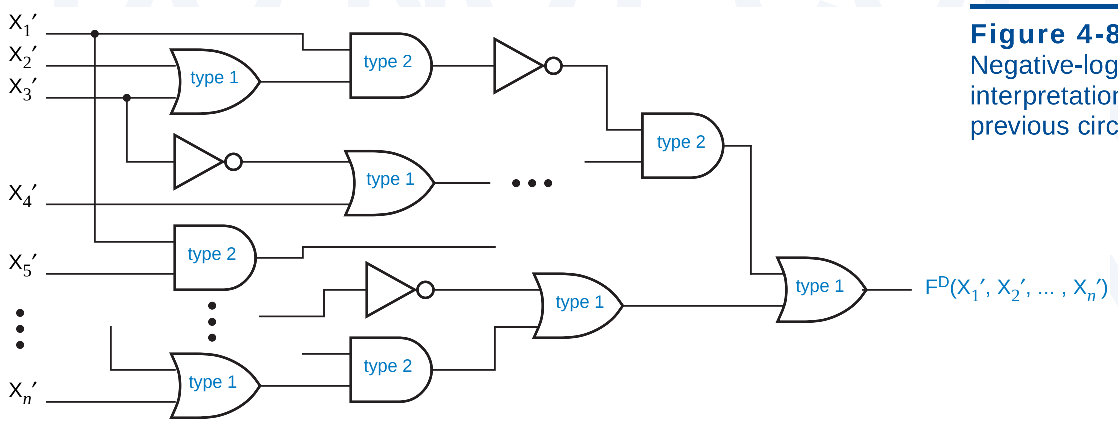


Figure 4-8
Negative-logic interpretation of the previous circuit.

By complementing both sides, we get the generalized DeMorgan's theorem:

$$[F(X_1,X_2,\ldots,X_n)]' = F^D(X_1',X_2',\ldots,X_n')$$

Amazing!

So, we have seen that duality is the basis for the generalized DeMorgan's theorem. Going forward, duality will halve the number of methods you must learn to manipulate and simplify logic functions.

4.1.6 Standard Representations of Logic Functions

Before moving on to analysis and synthesis of combinational logic functions we'll introduce some necessary nomenclature and notation.

truth table

The most basic representation of a logic function is the *truth table*. Similar in philosophy to the perfect-induction proof method, this brute-force representation simply lists the output of the circuit for every possible input combination. Traditionally, the input combinations are arranged in rows in ascending binary counting order, and the corresponding output values are written in a column next to the rows. The general structure of a 3-variable truth table is shown below in Table 4-4.

Table 4-4t General truth table structure for a 3-variable logic function, F(X,Y,Z).

Row	X	Y	Z	F
0	0	0	0	F(0,0,0)
1	0	0	1	F(0,0,1)
2	0	1	0	F(0,1,0)
3	0	1	1	F(0,1,1)
4	1	0	0	F(1,0,0)
5	1	0	1	F(1,0,1)
6	1	1	0	F(1,1,0)
7	1	1	1	F(1,1,1)

The rows are numbered 0–7 corresponding to the binary input combinations, but this numbering is not an essential part of the truth table. The truth table for a particular 3-variable logic function is shown in Table 4-5. Each distinct pattern of 0s and 1s in the output column yields a different logic function; there are 2^8 such patterns. Thus, the logic function in Table 4-5 is one of 2^8 different logic functions of three variables.

The truth table for an n-variable logic function has 2^n rows. Obviously, truth tables are practical to write only for logic functions with a small number of variables, say, 10 for students and about 4–5 for everyone else.

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Table 4-5 Truth table for a particular 3-variable logic function, F(X,Y,Z).

The information contained in a truth table can also be conveyed algebraically. To do so, we first need some definitions:

- A *literal* is a variable or the complement of a variable. Examples: X, Y, X′, Y′. *literal*
- A *product term* is a single literal or a logical product of two or more literals. Examples: Z', $W \cdot X \cdot Y$, $X \cdot Y' \cdot Z$, $W' \cdot Y' \cdot Z$. *product term*
- A *sum-of-products expression* is a logical sum of product terms. Example: $Z' + W \cdot X \cdot Y + X \cdot Y' \cdot Z + W' \cdot Y' \cdot Z$. *sum-of-products expression*
- A *sum term* is a single literal or a logical sum of two or more literals. Examples: Z', $W + X + Y$, $X + Y' + Z$, $W' + Y' + Z$. *sum term*
- A *product-of-sums expression* is a logical product of sum terms. Example: $Z' \cdot (W + X + Y) \cdot (X + Y' + Z) \cdot (W' + Y' + Z)$. *product-of-sums expression*
- A *normal term* is a product or sum term in which no variable appears more than once. A nonnormal term can always be simplified to a constant or a normal term using one of theorems T3, T3′, T5, or T5′. Examples of non-normal terms: $W \cdot X \cdot X \cdot Y'$, $W + W + X' + Y$, $X \cdot X' \cdot Y$. Examples of normal terms: $W \cdot X \cdot Y'$, $W + X' + Y$. *normal term*
- An *n*-variable *minterm* is a normal product term with *n* literals. There are 2^n such product terms. Examples of 4-variable minterms: $W' \cdot X' \cdot Y' \cdot Z'$, $W \cdot X \cdot Y' \cdot Z$, $W' \cdot X' \cdot Y \cdot Z'$. *minterm*
- An *n*-variable *maxterm* is a normal sum term with n literals. There are 2^n such sum terms. Examples of 4-variable maxterms: $W' + X' + Y' + Z'$, $W + X' + Y' + Z$, $W' + X' + Y + Z'$. *maxterm*

There is a close correspondence between the truth table and minterms and maxterms. A minterm can be defined as a product term that is 1 in exactly one row of the truth table. Similarly, a maxterm can be defined as a sum term that is 0 in exactly one row of the truth table. Table 4-6 shows this correspondence for a 3-variable truth table.

Table 4-6 Minterms and maxterms for a 3-variable logic function, F(X,Y,Z).

Row	X	Y	Z	F	***Minterm***	***Maxterm***
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	F(1,1,0)	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	F(1,1,1)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

minterm number
minterm i

An *n*-variable minterm can be represented by an *n*-bit integer, the *minterm number*. We'll use the name *minterm i* to denote the minterm corresponding to row *i* of the truth table. In minterm *i*, a particular variable appears complemented if the corresponding bit in the binary representation of *i* is 0; otherwise, it is uncomplemented. For example, row 5 has binary representation 101 and the corresponding minterm is $X \cdot Y' \cdot Z$. As you might expect, the correspondence for maxterms is just the opposite: in *maxterm i*, a variable appears complemented if the corresponding bit in the binary representation of *i* is 1. Thus, maxterm 5 (101) is $X' + Y + Z'$.

maxterm i

Based on the correspondence between the truth table and minterms, we can easily create an algebraic representation of a logic function from its truth table. The *canonical sum* of a logic function is a sum of the minterms corresponding to truth-table rows (input combinations) for which the function produces a 1 output. For example, the canonical sum for the logic function in Table 4-5 on page 205 is

canonical sum

$$\begin{aligned} F &= \Sigma_{X,Y,Z}(0,3,4,6,7) \\ &= X' \cdot Y' \cdot Z' + X' \cdot Y \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z' + X \cdot Y \cdot Z \end{aligned}$$

minterm list

Here, the notation $\Sigma_{X,Y,Z}(0,3,4,6,7)$ is a *minterm list* and means "the sum of minterms 0, 3, 4, 6, and 7 with variables X, Y, and Z." The minterm list is also known as the *on-set* for the logic function. You can visualize that each minterm "turns on" the output for exactly one input combination. Any logic function can be written as a canonical sum.

on-set

canonical product

The *canonical product* of a logic function is a product of the maxterms corresponding to input combinations for which the function produces a 0 output. For example, the canonical product for the logic function in Table 4-5 is

$$\begin{aligned} F &= \Pi_{X,Y,Z}(1,2,5) \\ &= (X + Y + Z') \cdot (X + Y' + Z) \cdot (X' + Y + Z') \end{aligned}$$

Here, the notation $\prod_{X,Y,Z}(1,2,5)$ is a *maxterm list* and means "the product of maxterms 1, 2, and 5 with variables X, Y, and Z." The maxterm list is also known as the *off-set* for the logic function. You can visualize that each maxterm "turns off" the output for exactly one input combination. Any logic function can be written as a canonical product.

maxterm list

off-set

It's easy to convert between a minterm list and a maxterm list. For a function of n variables, the possible minterm and maxterm numbers are in the set $\{0, 1, \ldots, 2^n-1\}$; a minterm or maxterm list contains a subset of these numbers. To switch between list types, take the set complement, for example,

$$\begin{aligned}\Sigma_{A,B,C}(0,1,2,3) &= \prod_{A,B,C}(4,5,6,7)\\ \Sigma_{X,Y}(1) &= \prod_{X,Y}(0,2,3)\\ \Sigma_{W,X,Y,Z}(0,1,2,3,5,7,11,13) &= \prod_{W,X,Y,Z}(4,6,8,9,10,12,14,15)\end{aligned}$$

We have now learned five possible representations for a combinational logic function:

1. A truth table.
2. An algebraic sum of minterms, the canonical sum.
3. A minterm list using the Σ notation.
4. An algebraic product of maxterms, the canonical product.
5. A maxterm list using the Π notation.

Each one of these representations specifies exactly the same information; given any one of them, we can derive the other four using a simple mechanical process.

4.2 Combinational Circuit Analysis

We analyze a combinational logic circuit by obtaining a formal description of its logic function. Once we have a description of the logic function, a number of other operations are possible:

- We can determine the behavior of the circuit for various input combinations.
- We can manipulate an algebraic description to suggest different circuit structures for the logic function.
- We can transform an algebraic description into a standard form corresponding to an available circuit structure. For example, a sum-of-products expression corresponds directly to the circuit structure used in PLDs (programmable logic devices).
- We can use an algebraic description of the circuit's functional behavior in the analysis of a larger system that includes the circuit.

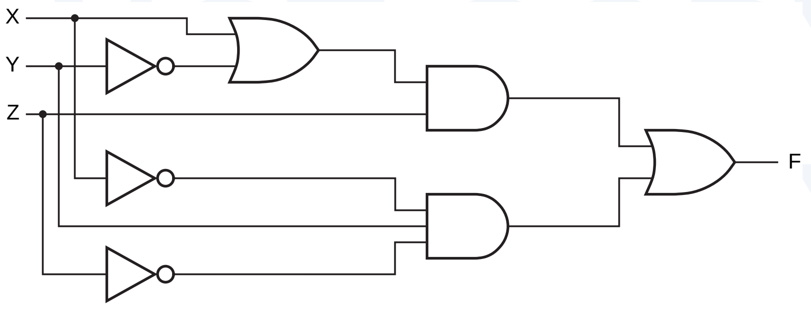
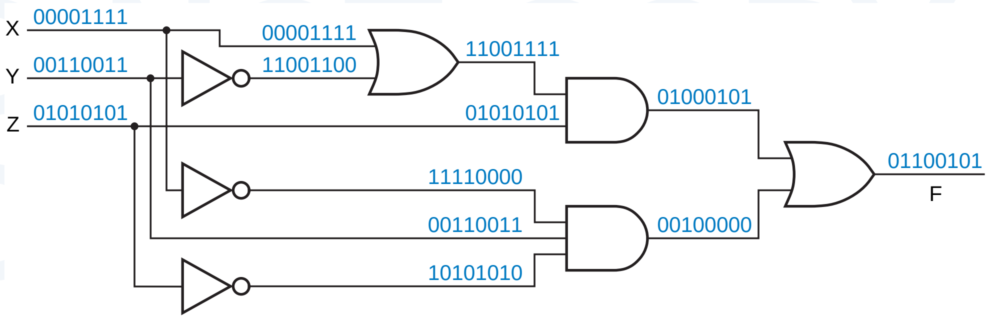


Figure 4-9
A three-input, one-output logic circuit.

Given a logic diagram for a combinational circuit, such as Figure 4-9, there are a number of ways to obtain a formal description of the circuit's function. The most primitive functional description is the truth table.

Using only the basic axioms of switching algebra, we can obtain the truth table of an n-input circuit by working our way through all 2^n input combinations. For each input combination, we determine all of the gate outputs produced by that input, propagating information from the circuit inputs to the circuit outputs. Figure 4-10 applies this "exhaustive" technique to our example circuit. Written on each signal line in the circuit is a sequence of eight logic values, the values present on that line when the circuit inputs XYZ are 000, 001, …, 111. The truth table can be written by transcribing the output sequence of the final OR gate, as

Figure 4-10 Gate outputs created by all input combinations.



A LESS EXHAUSTING WAY TO GO

You can easily obtain the results in Figure 4-10 with typical logic design tools that include a logic simulator. First, you draw the schematic. Then, you apply the outputs of a 3-bit binary counter to the X, Y, and Z inputs. (Most simulators have such counter outputs built-in for just this sort of exercise.) The counter repeatedly cycles through the eight possible input combinations, in the same order that we've shown in the figure. The simulator allows you to graph the resulting signal values at any point in the schematic, including the intermediate points as well as the output.

Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Table 4-7
Truth table for the logic circuit of Figure 4-9.

shown in Table 4-7. Once we have the truth table for the circuit, we can also directly write a logic expression—the canonical sum or product—if we wish.

The number of input combinations of a logic circuit grows exponentially with the number of inputs, so the exhaustive approach can quickly become exhausting. Instead, we normally use an algebraic approach whose complexity is more linearly proportional to the size of the circuit. The method is simple—we build up a parenthesized logic expression corresponding to the logic operators and structure of the circuit. We start at the circuit inputs and propagate expressions through gates toward the output. Using the theorems of switching algebra, we may simplify the expressions as we go, or we may defer all algebraic manipulations until an output expression is obtained.

Figure 4-11 applies the algebraic technique to our example circuit. The output function is given on the output of the final OR gate:

$$F = ((X+Y') \cdot Z) + (X' \cdot Y \cdot Z')$$

No switching-algebra theorems were used in obtaining this expression. However, we can use theorems to transform this expression into another form. For example, a sum of products can be obtained by "multiplying out":

$$F = X \cdot Z + Y' \cdot Z + X' \cdot Y \cdot Z'$$

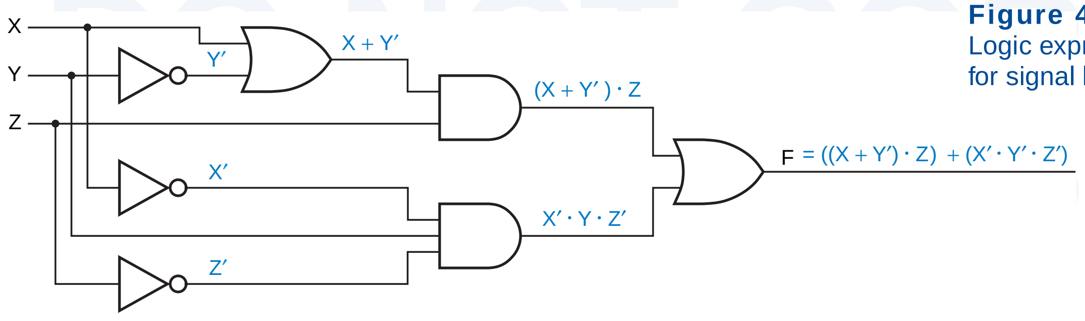


Figure 4-11
Logic expressions for signal lines.

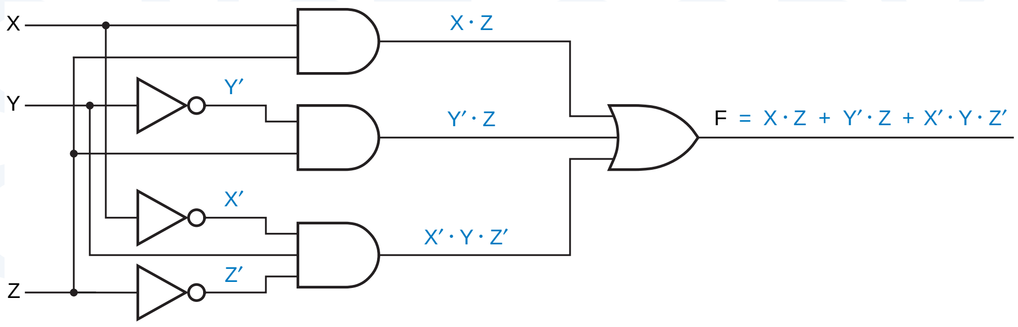


Figure 4-12 Two-level AND-OR circuit.

The new expression corresponds to a different circuit for the same logic function, as shown in Figure 4-12.

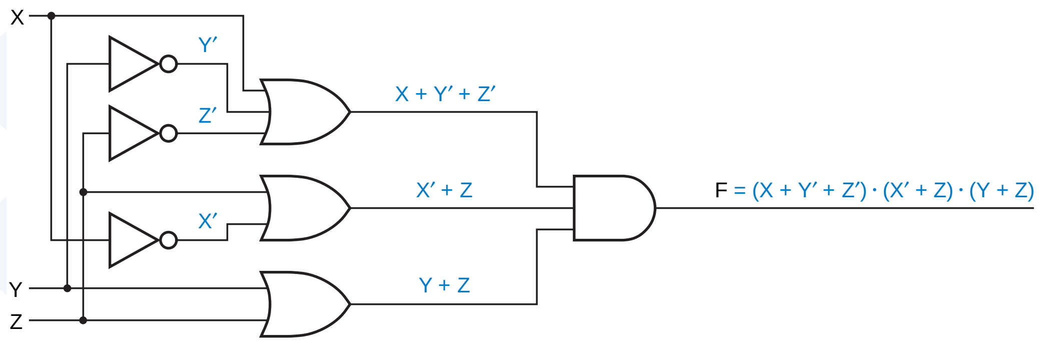
Similarly, we can "add out" the original expression to obtain a product of sums:

$$\begin{aligned} F &= ((X + Y') \cdot Z) + (X' \cdot Y \cdot Z') \\ &= (X + Y' + X') \cdot (X + Y' + Y) \cdot (X + Y' + Z') \cdot (Z + X') \cdot (Z + Y) \cdot (Z + Z') \\ &= 1 \cdot 1 \cdot (X + Y' + Z') \cdot (X' + Z) \cdot (Y + Z) \cdot 1 \\ &= (X + Y' + Z') \cdot (X' + Z) \cdot (Y + Z) \end{aligned}$$

The corresponding logic circuit is shown in Figure 4-13.

Our next example of algebraic analysis uses a circuit with NAND and NOR gates, shown in Figure 4-14. This analysis is a little messier than the previous example, because each gate produces a complemented subexpression, not just a simple sum or product. However, the output expression can be simplified by repeated application of the generalized DeMorgan's theorem:

Figure 4-13 Two-level OR-AND circuit.



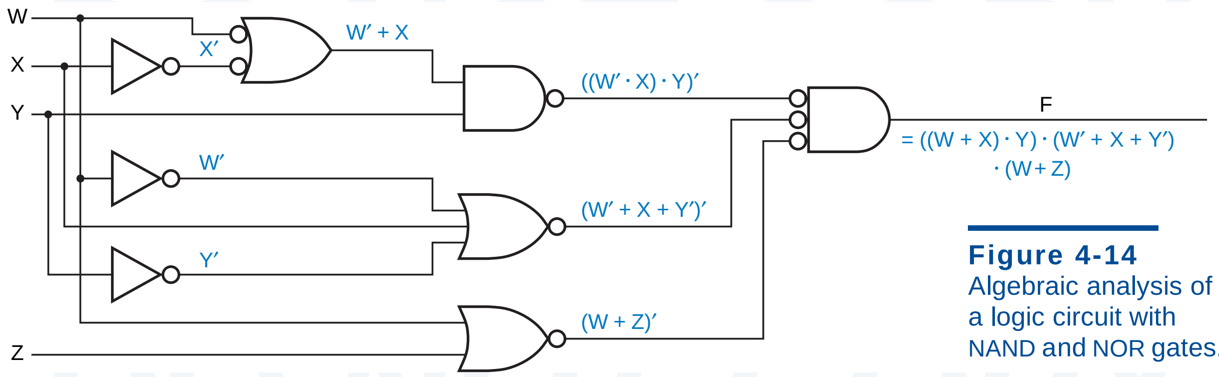


Figure 4-14 Algebraic analysis of a logic circuit with NAND and NOR gates.

$$\begin{aligned}
F &= [((W \cdot X')' \cdot Y)' + (W' + X + Y')' + (W + Z)']' \\
&= ((W' + X)' + Y')' \cdot (W \cdot X' \cdot Y)' \cdot (W' \cdot Z')' \\
&= ((W \cdot X')' \cdot Y) \cdot (W' + X + Y') \cdot (W + Z) \\
&= ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z)
\end{aligned}$$

Quite often, DeMorgan's theorem can be applied *graphically* to simplify algebraic analysis. Recall from Figures 4-3 and 4-4 that NAND and NOR gates each have two equivalent symbols. By judiciously redrawing Figure 4-14, we make it possible to cancel out some of the inversions during the analysis by using theorem T4 $[(X')' = X]$, as shown in Figure 4-15. This manipulation leads us to a simplified output expression directly:

$$F = ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z)$$

Figures 4-14 and 4-15 were just two different ways of drawing the same physical logic circuit. However, when we simplify a logic expression using the theorems of switching algebra, we get an expression corresponding to a different physical circuit. For example, the simplified expression above corresponds to the circuit of Figure 4-16, which is physically different from the one in the previous two figures. Furthermore, we could multiply out and add out the

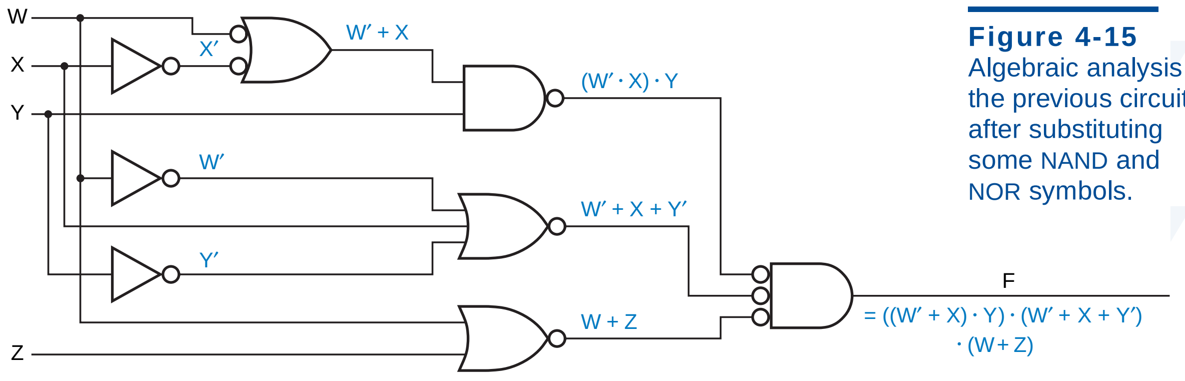


Figure 4-15 Algebraic analysis of the previous circuit after substituting some NAND and NOR symbols.

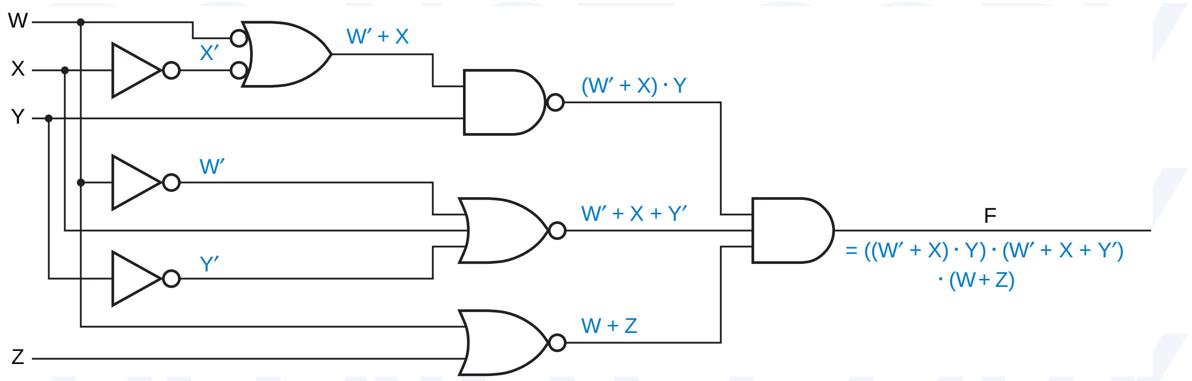
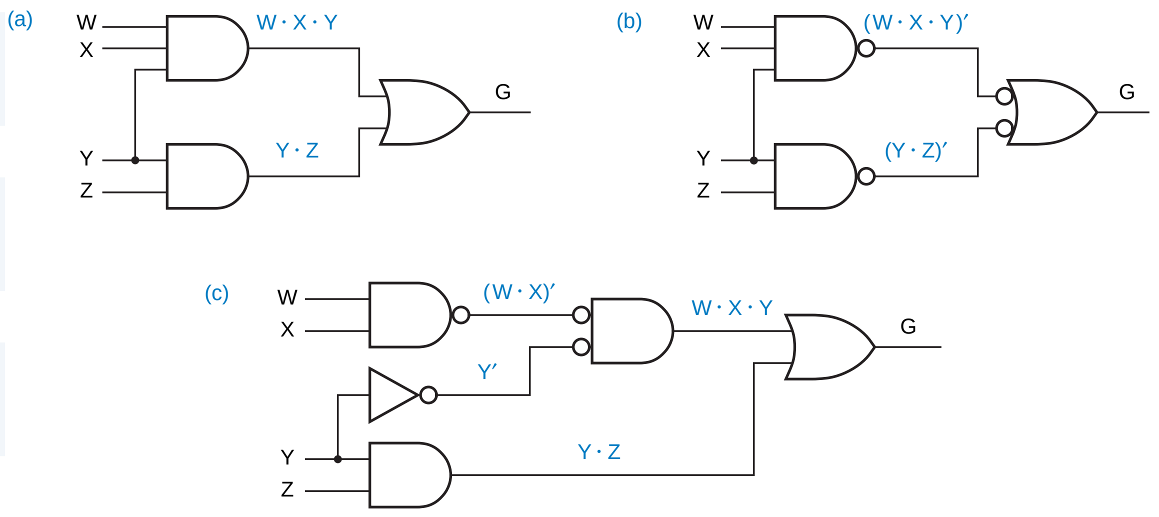


Figure 4-16 A different circuit for same logic function.

expression to obtain sum-of-products and product-of-sums expressions corresponding to two more physically different circuits for the same logic function.

Although we used logic expressions above to convey information about the physical structure of a circuit, we don't always do this. For example, we might use the expression G(W, X, Y, Z) = W · X · Y + Y · Z to describe any one of the circuits in Figure 4-17. Normally, the only sure way to determine a circuit's structure is to look at its schematic drawing. However, for certain restricted classes of circuits, structural information can be inferred from logic expressions. For example, the circuit in (a) could be described without reference to the drawing as "a two-level AND-OR circuit for W · X · Y + Y · Z," while the circuit in (b) could be described as "a two-level NAND-NAND circuit for W · X · Y + Y · Z."

Figure 4-17 Three circuits for G(W, X, Y, Z) = W · X ·Y + Y · Z: (a) two-level AND-OR; (b) two-level NAND-NAND; (c) ad hoc.



4.3 Combinational Circuit Synthesis

4.3.1 Circuit Descriptions and Designs

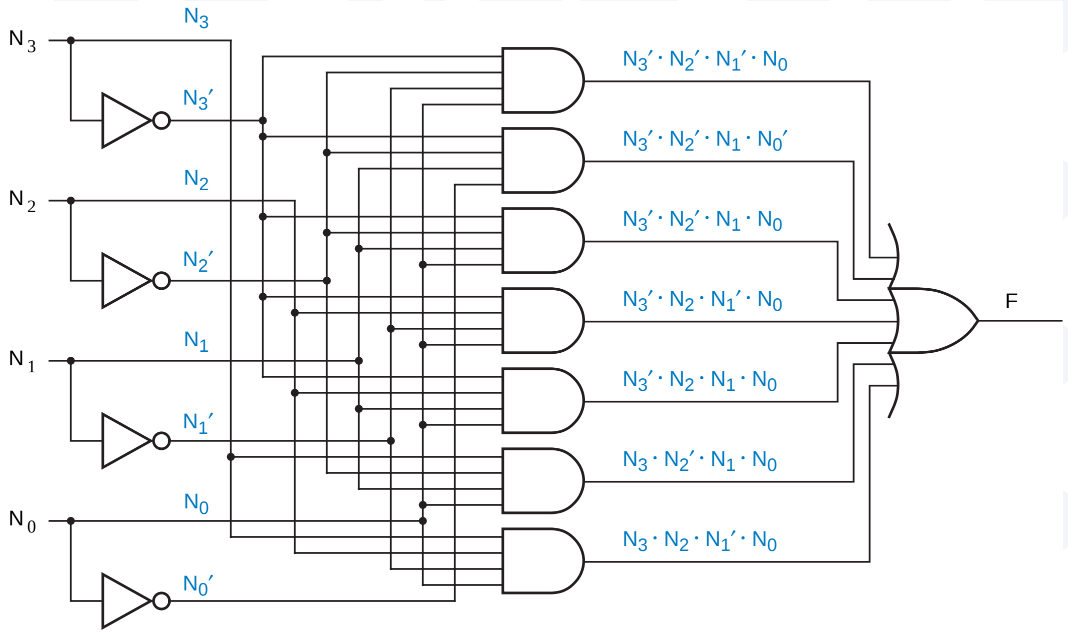
What is the starting point for designing combinational logic circuits? Usually, we are given a word description of a problem or we develop one ourselves. Occasionally, the description is a list of input combinations for which a signal should be on or off, the verbal equivalent of a truth table or the Σ or $\prod$ notation introduced previously. For example, the description of a 4-bit prime-number detector might be, "Given a 4-bit input combination $N = N_3N_2N_1N_0$, this function produces a 1 output for N = 1, 2, 3, 5, 7, 11, 13, and 0 otherwise." A logic function described in this way can be designed directly from the canonical sum or product expression. For the prime-number detector, we have

$$\begin{aligned} F &= \Sigma_{N_3,N_2,N_1,N_0}(1, 2, 3, 5, 7, 11, 13) \\ &= N_3'\cdot N_2'\cdot N_1'\cdot N_0 + N_3'\cdot N_2'\cdot N_1\cdot N_0' + N_3'\cdot N_2'\cdot N_1\cdot N_0 + N_3'\cdot N_2'\cdot N_1'\cdot N_0 \\ &\quad + N_3'\cdot N_2\cdot N_1\cdot N_0 + N_3\cdot N_2'\cdot N_1\cdot N_0 + N_3\cdot N_2\cdot N_1'\cdot N_0 \end{aligned}$$

The corresponding circuit is shown in Figure 4-18.

More often, we describe a logic function using the English-language connectives "and," "or," and "not." For example, we might describe an alarm circuit by saying, "The ALARM output is 1 if the PANIC input is 1, or if the ENABLE input is 1, the EXITING input is 0, and the house is not secure; the house is secure

Figure 4-18 Canonical-sum design for 4-bit prime-number detector.



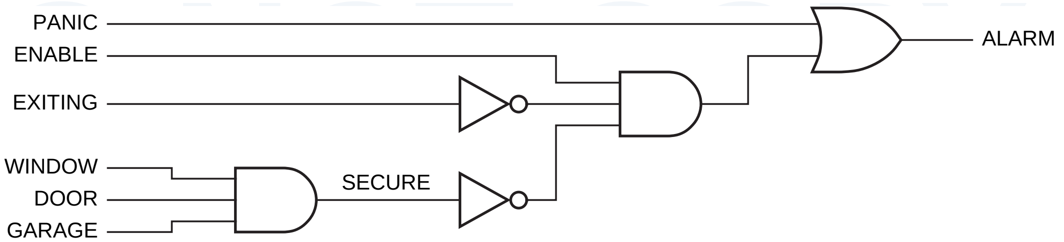


Figure 4-19 Alarm circuit derived from logic expression.

if the WINDOW, DOOR, and GARAGE inputs are all 1." Such a description can be translated directly into algebraic expressions:

$$\text{ALARM} = \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot \text{SECURE}'$$

$$\text{SECURE} = \text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE}$$

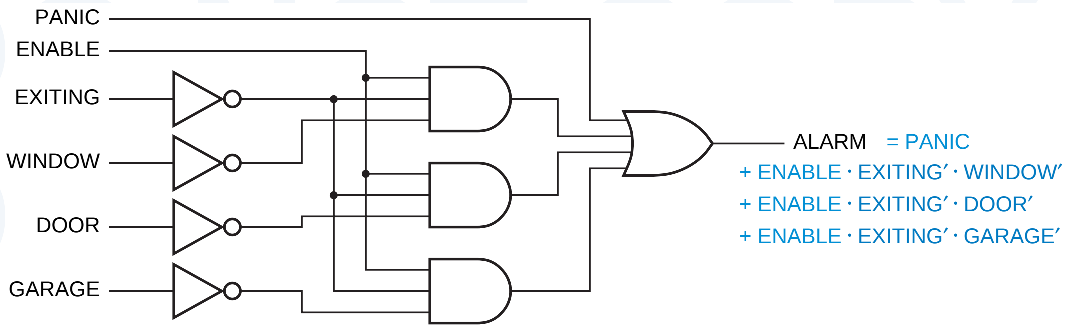
$$\text{ALARM} = \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot (\text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE})'$$

Notice that we used the same method in switching algebra as in ordinary algebra to formulate a complicated expression—we defined an auxiliary variable SECURE to simplify the first equation, developed an expression for SECURE, and used substitution to get the final expression. We can easily draw a circuit using AND, OR, and NOT gates that realizes the final expression, as shown in Figure 4-19. A circuit *realizes* ["makes real"] an expression if its output function equals that expression, and the circuit is called a *realization* of the function.

realize
realization

Once we have an expression, any expression, for a logic function, we can do other things besides building a circuit directly from the expression. We can manipulate the expression to get different circuits. For example, the ALARM expression above can be multiplied out to get the sum-of-products circuit in Figure 4-20. Or, if the number of variables is not too large, we can construct the truth table for the expression and use any of the synthesis methods that apply to truth tables, including the canonical sum or product method described earlier and the minimization methods described later.

Figure 4-20 Sum-of-products version of alarm circuit.


PANIC
ENABLE
EXITING
WINDOW
DOOR
GARAGE
ALARM = PANIC
+ ENABLE · EXITING′ · WINDOW′
+ ENABLE · EXITING′ · DOOR′
+ ENABLE · EXITING′ · GARAGE′

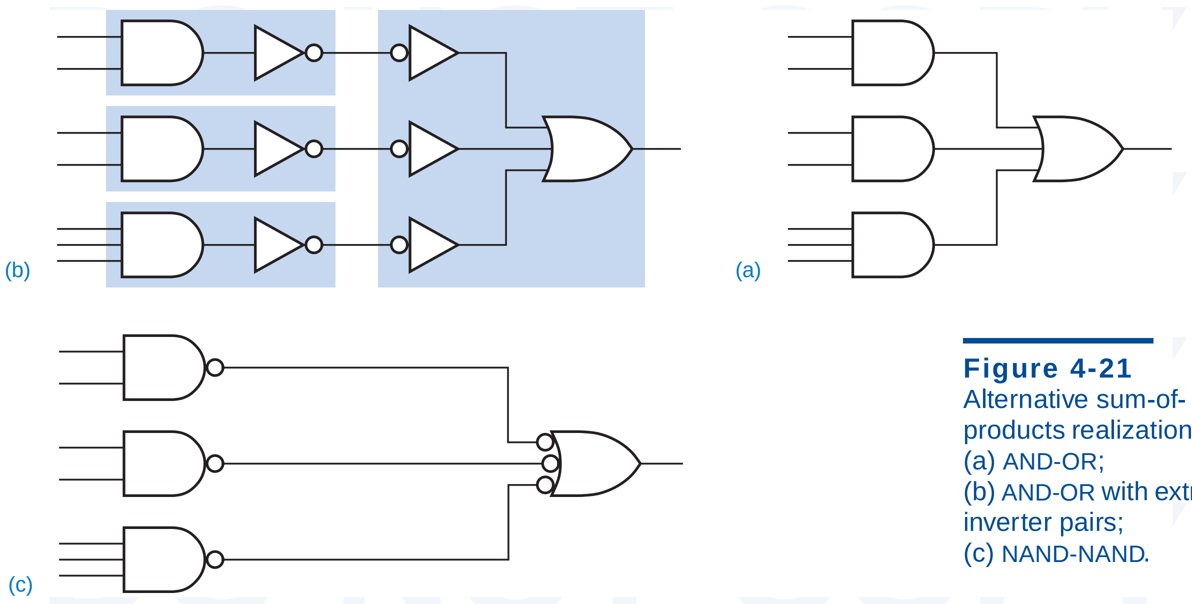


Figure 4-21
Alternative sum-of-products realizations:
(a) AND-OR;
(b) AND-OR with extra inverter pairs;
(c) NAND-NAND.

In general, it's easier to describe a circuit in words using logical connectives and to write the corresponding logic expressions than it is to write a complete truth table, especially if the number of variables is large. However, sometimes we have to work with imprecise word descriptions of logic functions, for example, "The ERROR output should be 1 if the GEARUP, GEARDOWN, and GEARCHECK inputs are inconsistent." In this situation, the truth-table approach is best because it allows us to determine the output required for every input combination, based on our knowledge and understanding of the problem environment (e.g., the brakes cannot be applied unless the gear is down).

4.3.2 Circuit Manipulations

The design methods that we've described so far use AND, OR, and NOT gates. We might like to use NAND and NOR gates, too—they're faster than ANDs and ORs in most technologies. However, most people don't develop logical propositions in terms of NAND and NOR connectives. That is, you probably wouldn't say, "I won't date you if you're not clean or not wealthy and also you're not smart or not friendly." It would be more natural for you to say, "I'll date you if you're clean and wealthy, or if you're smart and friendly." So, given a "natural" logic expression, we need ways to translate it into other forms.

We can translate any logic expression into an equivalent sum-of-products expression, simply by multiplying it out. As shown in Figure 4-21(a), such an expression may be realized directly with AND and OR gates. The inverters required for complemented inputs are not shown.

As shown in Figure 4-21(b), we may insert a pair of inverters between each AND-gate output and the corresponding OR-gate input in a two-level AND-OR

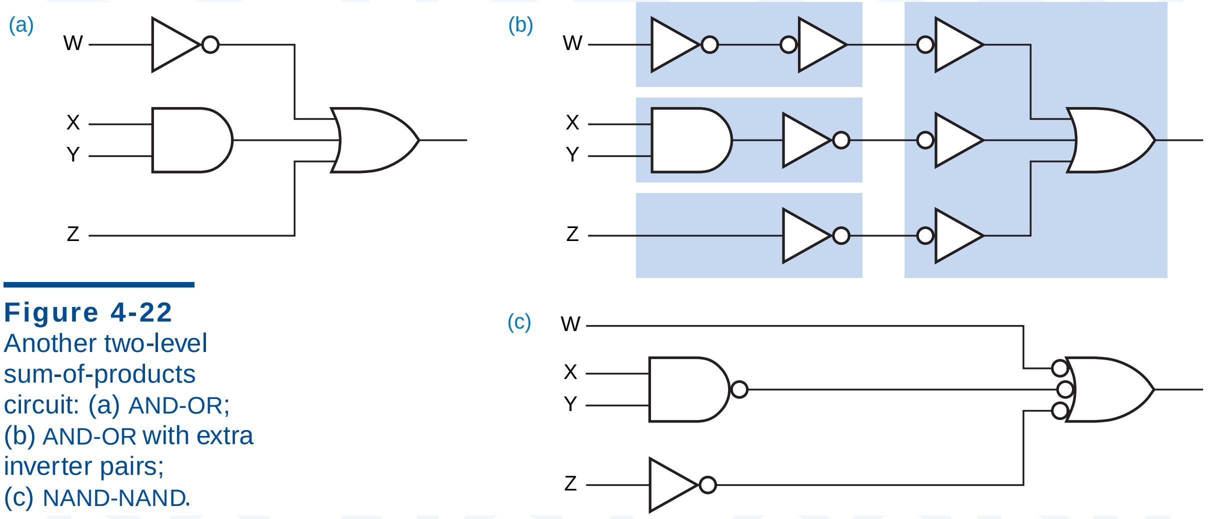
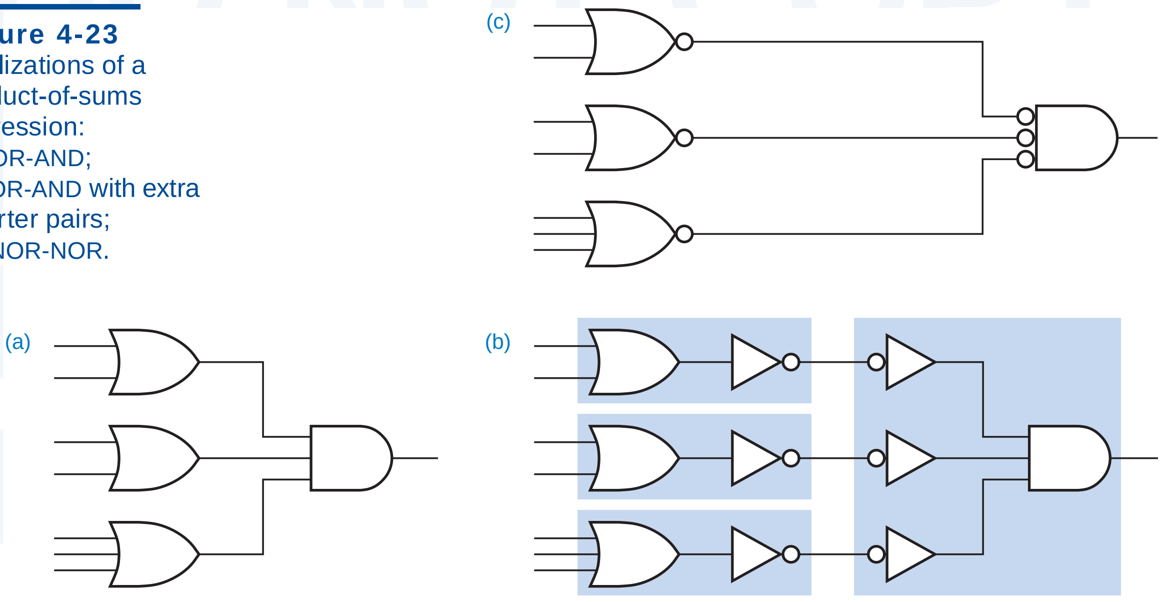


Figure 4-22
Another two-level sum-of-products circuit: (a) AND-OR; (b) AND-OR with extra inverter pairs; (c) NAND-NAND.

circuit. According to theorem T4, these inverters have no effect on the output function of the circuit. In fact, we've drawn the second inverter of each pair with its inversion bubble on its input to provide a graphical reminder that the inverters cancel. However, if these inverters are absorbed into the AND and OR gates, we wind up with AND-NOT gates at the first level and a NOT-OR gate at the second level. These are just two different symbols for the same type of gate—a NAND gate. Thus, a two-level AND-OR *circuit* may be converted to a two-level NAND-NAND *circuit* simply by substituting gates.

AND-OR circuit
NAND-NAND circuit

Figure 4-23
Realizations of a product-of-sums expression: (a) OR-AND; (b) OR-AND with extra inverter pairs; (c) NOR-NOR.



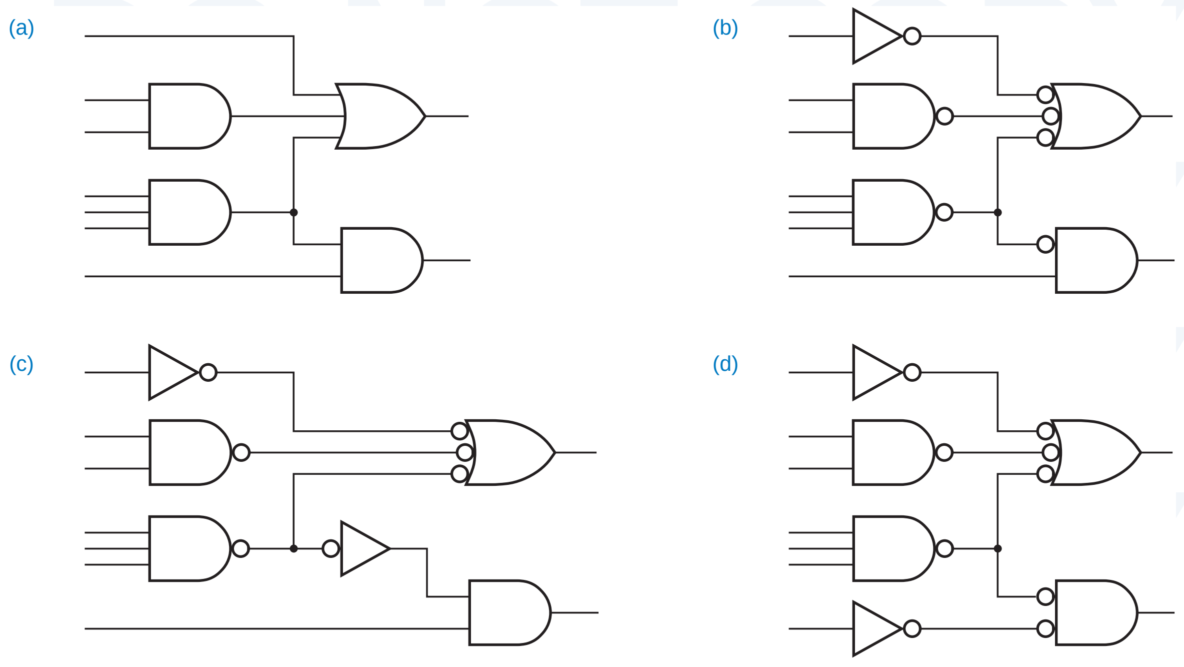
If any product terms in the sum-of-products expression contain just a single literal, then we may gain or lose inverters in the transformation from AND-OR to NAND-NAND. For example, Figure 4-22 is an example where an inverter on the W input is no longer needed, but an inverter must be added to the Z input.

We have shown that any sum-of-products expression can be realized in either of two ways—as an AND-OR circuit or as a NAND-NAND circuit. The dual of this statement is also true: any product-of-sums expression can be realized as an OR-AND *circuit* or as a NOR-NOR *circuit*. Figure 4-23 shows an example. Any logic expression can be translated into an equivalent product-of-sums expression by adding it out, and hence has both OR-AND and NOR-NOR circuit realizations.

OR-AND circuit
NOR-NOR circuit

The same kind of manipulations can be applied to arbitrary logic circuits. For example, Figure 4-24(a) shows a circuit built from AND and OR gates. After adding pairs of inverters, we obtain the circuit in (b). However, one of the gates, a 2-input AND gate with a single inverted input, is not a standard type. We can use a discrete inverter as shown in (c) to obtain a circuit that uses only standard gate types—NAND, AND, and inverters. Actually, a better way to use the inverter is shown in (d); one level of gate delay is eliminated, and the bottom gate becomes a NOR instead of AND. In most logic technologies, inverting gates like NAND and NOR are faster than noninverting gates like AND and OR.

Figure 4-24 Logic-symbol manipulations: (a) original circuit; (b) transformation with a nonstandard gate; (c) inverter used to eliminate nonstandard gate; (d) preferred inverter placement.



WHY MINIMIZE? Minimization is an important step in both ASIC design and in design PLDs. Extra gates and gate inputs require more area in an ASIC chip, and thereby increase cost. The number of gates in a PLD is fixed, so you might think that extra gates are free—and they are, until you run out of them and have to upgrade to a bigger, slower, more expensive PLD. Fortunately, most software tools for both ASIC and PLD design have a minimization program built in. The purpose of Sections 4.3.3 through 4.3.8 is to give you a feel for how minimization works.

4.3.3 Combinational Circuit Minimization

It's often uneconomical to realize a logic circuit directly from the first logic expression that pops into your head. Canonical sum and product expressions are especially expensive because the number of possible minterms or maxterms (and hence gates) grows exponentially with the number of variables. We *minimize* a combinational circuit by reducing the number and size of gates that are needed to build it.

minimize

The traditional combinational circuit minimization methods that we'll study have as their starting point a truth table or, equivalently, a minterm list or maxterm list. If we are given a logic function that is not expressed in this form, then we must convert it to an appropriate form before using these methods. For example, if we are given an arbitrary logic expression, then we can evaluate it for every input combination to construct the truth table. The minimization methods reduce the cost of a two-level AND-OR, OR-AND, NAND-NAND, or NOR-NOR circuit in three ways:

1. By minimizing the number of first-level gates.
2. By minimizing the number of inputs on each first-level gate.
3. By minimizing the number of inputs on the second-level gate. This is actually a side effect of the first reduction.

However, the minimization methods do not consider the cost of input inverters; they assume that both true and complemented versions of all input variables are available. While this is not always the case in gate-level or ASIC design, it's very appropriate for PLD-based design; PLDs have both true and complemented versions of all input variables available "for free."

Most minimization methods are based on a generalization of the combining theorems, T10 and T10′:

$$\text{given product term} \cdot \mathrm{Y} + \text{given product term} \cdot \mathrm{Y}' = \text{given product term}$$
$$\text{given sum term} + \mathrm{Y}) \cdot \text{given sum term} + \mathrm{Y}') = \text{given sum term}$$

That is, if two product or sum terms differ only in the complementing or not of one variable, we can combine them into a single term with one less variable. So we save one gate and the remaining gate has one fewer input.

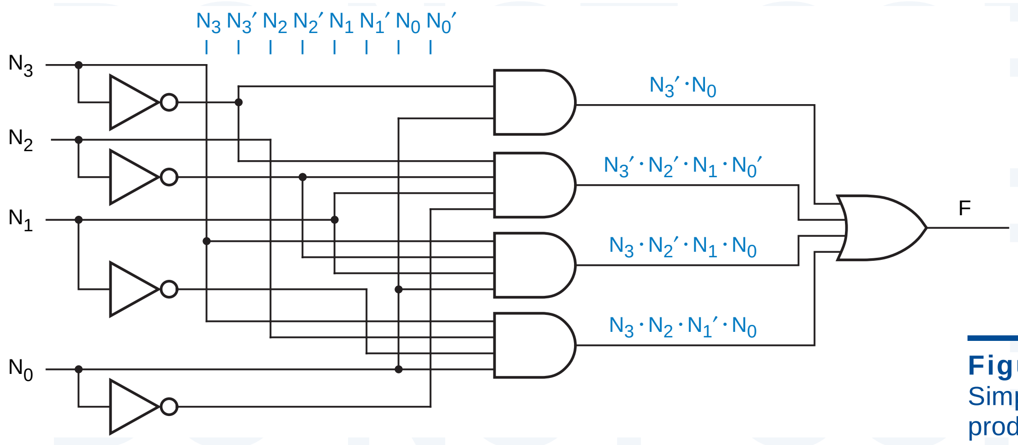


Figure 4-25 Simplified sum-of-products realization for 4-bit prime-number detector.

We can apply this algebraic method repeatedly to combine minterms 1, 3, 5, and 7 of the prime-number detector shown in Figure 4-18 on page 213:

$$\begin{aligned}
F &= \Sigma_{N_3,N_2,N_1,N_0}(1, 3, 5, 7, 2, 11, 13) \\
&= N_3{}'\cdot N_2{}'N_1{}'N_0 + N_3{}'\cdot N_2{}'\cdot N_1\cdot N_0 + N_3{}'\cdot N_2\cdot N_1{}'\cdot N_0 + N_3{}'\cdot N_2\cdot N_1\cdot N_0 + \ldots \\
&= (N_3{}'\cdot N_2{}'\cdot N_1{}'\cdot N_0 + N_3{}'\cdot N_2{}'\cdot N_1\cdot N_0) + (\cdot N_3{}'\cdot N_2\cdot N_1{}'\cdot N_0 + N_3{}'\cdot N_2\cdot N_1\cdot N_0) + \ldots \\
&= N_3{}'N_2{}'\cdot N_0 + N_3{}'\cdot N_2\cdot N_0 + \ldots \\
&= N_3{}'\cdot N_0 + \ldots
\end{aligned}$$

The resulting circuit is shown in Figure 4-25; it has three fewer gates and one of the remaining gates has two fewer inputs.

If we had worked a little harder on the preceding expression, we could have saved a couple more first-level gate inputs, though not any gates. It's difficult to find terms that can be combined in a jumble of algebraic symbols. In the next subsection, we'll begin to explore a minimization method that is more fit for human consumption. Our starting point will be the graphical equivalent of a truth table.

4.3.4 Karnaugh Maps

Karnaugh map

A *Karnaugh map* is a graphical representation of a logic function's truth table. Figure 4-26 shows Karnaugh maps for logic functions of 2, 3, and 4 variables. The map for an *n*-input logic function is an array with 2^n cells, one for each possible input combination or minterm.

The rows and columns of a Karnaugh map are labeled so that the input combination for any cell is easily determined from the row and column headings for that cell. The small number inside each cell is the corresponding minterm number in the truth table, assuming that the truth table inputs are labeled alphabetically from left to right (e.g., X, Y, Z) and the rows are numbered in binary

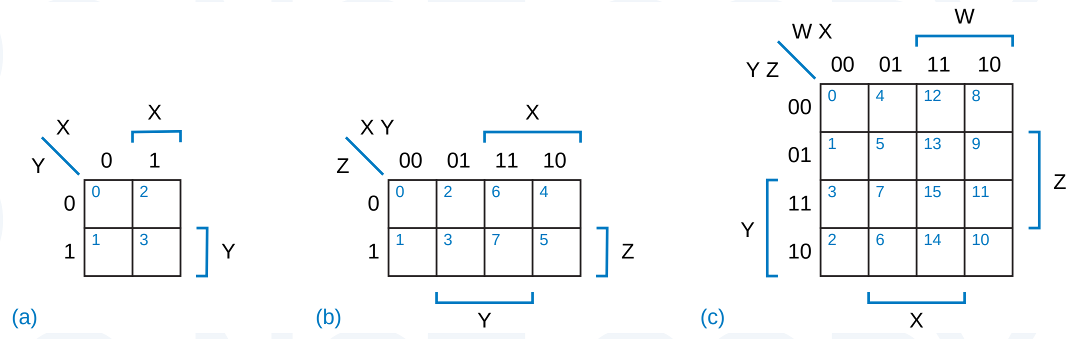


Figure 4-26 Karnaugh maps: (a) 2-variable; (b) 3-variable; (c) 4-variable.

counting order, like all the examples in this text. For example, cell 13 in the 4-variable map corresponds to the truth-table row in which W X Y Z = 1101.

When we draw the Karnaugh map for a given function, each cell of the map contains the information from the like-numbered row of the function's truth table—a 0 if the function is 0 for that input combination, a 1 otherwise.

In this text, we use two redundant labelings for map rows and columns. For example, consider the 4-variable map in Figure 4-26(c). The columns are labeled with the four possible combinations of W and X, W X = 00, 01, 11, and 10. Similarly, the rows are labeled with the Y Z combinations. These labels give us all the information we need. However, we also use brackets to associate four regions of the map with the four variables. Each bracketed region is the part of the map in which the indicated variable is 1. Obviously, the brackets convey the same information that is given by the row and column labels.

When we draw a map by hand, it is much easier to draw the brackets than to write out all of the labels. However, we retain the labels in the text's Karnaugh maps as an additional aid to understanding. In any case, you must be sure to label the rows and columns in the proper order to preserve the correspondence between map cells and truth table row numbers shown in Figure 4-26.

To represent a logic function on a Karnaugh map, we simply copy 1s and 0s from the truth table or equivalent to the corresponding cells of the map. Figures 4-27(a) and (b) show the truth table and Karnaugh map for a logic function that we analyzed (beat to death?) in Section 4.2. From now on, we'll reduce the clutter in maps by copying only the 1s or the 0s, not both.

4.3.5 Minimizing Sums of Products

By now you must be wondering about the "strange" ordering of the row and column numbers in a Karnaugh map. There is a very important reason for this ordering—each cell corresponds to an input combination that differs from each of its immediately adjacent neighbors in only one variable. For example, cells 5 and 13 in the 4-variable map differ only in the value of W. In the 3- and

(a)

X	Y	Z	F
0	0	0	0
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

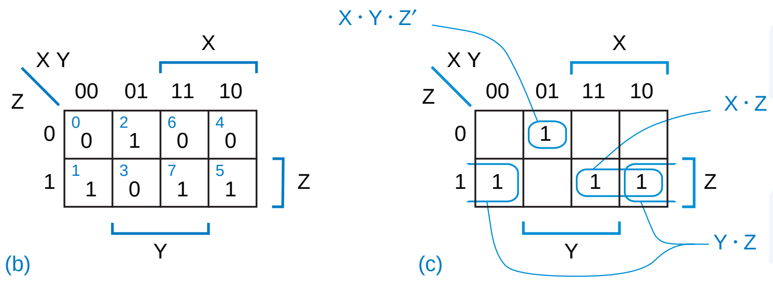


Figure 4-27 F = $\Sigma_{X,Y,Z}(1,2,5,7)$: (a) truth table; (b) Karnaugh map; (c) combining adjacent 1-cells.

4-variable maps, corresponding cells on the left/right or top/bottom borders are less obvious neighbors; for example, cells 12 and 14 in the 4-variable map are adjacent because they differ only in the value of Y.

Each input combination with a “1” in the truth table corresponds to a minterm in the logic function’s canonical sum. Since pairs of adjacent “1” cells in the Karnaugh map have minterms that differ in only one variable, the minterm pairs can be combined into a single product term using the generalization of theorem T10, term $\cdot$ Y + term $\cdot$ Y′ = term. Thus, we can use a Karnaugh map to simplify the canonical sum of a logic function.

For example, consider cells 5 and 7 in Figure 4-27(b), and their contribution to the canonical sum for this function:

$$\begin{aligned} F &= \ldots + X \cdot Y' \cdot Z + X \cdot Y \cdot Z \\ &= \ldots + (X \cdot Z) \cdot Y' + (X \cdot Z) \cdot Y \\ &= \ldots + X \cdot Z \end{aligned}$$

Remembering wraparound, we see that cells 1 and 5 in Figure 4-27(b) are also adjacent and can be combined:

$$\begin{aligned} F &= X' \cdot Y' \cdot Z + X \cdot Y' \cdot Z + \ldots \\ &= X' \cdot (Y' \cdot Z) + X \cdot (Y' \cdot Z) + \ldots \\ &= Y' \cdot Z + \ldots \end{aligned}$$

In general, we can simplify a logic function by combining pairs of adjacent 1-cells (minterms) whenever possible, and writing a sum of product terms that cover all of the 1-cells. Figure 4-27(c) shows the result for our example logic function. We circle a pair of 1s to indicate that the corresponding minterms are combined into a single product term. The corresponding AND-OR circuit is shown in Figure 4-28.

In many logic functions, the cell-combining procedure can be extended to combine more than two 1-cells into a single product term. For example, consider

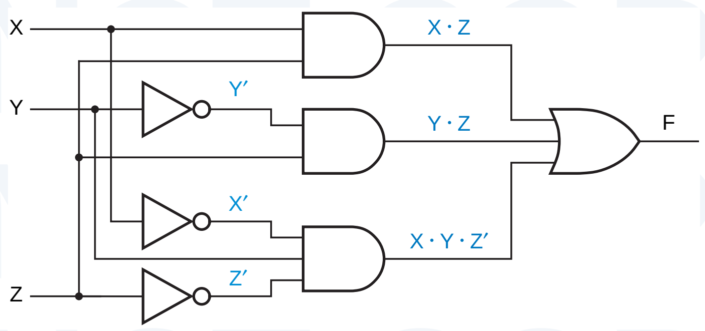


Figure 4-28
Minimized AND-OR circuit.

the canonical sum for the logic function $F = \Sigma_{X,Y,Z}(0, 1, 4, 5, 6)$. We can use the algebraic manipulations of the previous examples iteratively to combine four of the five minterms:

$$\begin{aligned} F &= X' \cdot Y' \cdot Z' + X' \cdot Y' \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y' \cdot Z + X \cdot Y \cdot Z' \\ &= [(X' \cdot Y') \cdot Z' + (X' \cdot Y') \cdot Z] + [(X \cdot Y') \cdot Z' + (X \cdot Y') \cdot Z] + X \cdot Y \cdot Z' \\ &= X' \cdot Y' + X \cdot Y' + X \cdot Y \cdot Z' \\ &= [X' \cdot (Y') + X \cdot (Y')] + X \cdot Y \cdot Z' \\ &= Y' + X \cdot Y \cdot Z' \end{aligned}$$

In general, 2^i 1-cells may be combined to form a product term containing $n - i$ literals, where n is the number of variables in the function.

A precise mathematical rule determines how 1-cells may be combined and the form of the corresponding product term:

- A set of 2^i 1-cells may be combined if there are i variables of the logic function that take on all 2^i possible combinations within that set, while the remaining $n - i$ variables have the same value throughout that set. The corresponding product term has $n - i$ literals, where a variable is complemented if it appears as 0 in all of the 1-cells, and uncomplemented if it appears as 1.

rectangular sets of 1s

Graphically, this rule means that we can circle *rectangular* sets of 2^n 1s, literally as well as figuratively stretching the definition of rectangular to account for wraparound at the edges of the map. We can determine the literals of the corresponding product terms directly from the map; for each variable we make the following determination:

- If a circle covers only areas of the map where the variable is 0, then the variable is complemented in the product term.
- If a circle covers only areas of the map where the variable is 1, then the variable is uncomplemented in the product term.
- If a circle covers both areas of the map where the variable is 0 and areas where it is 1, then the variable does not appear in the product term.

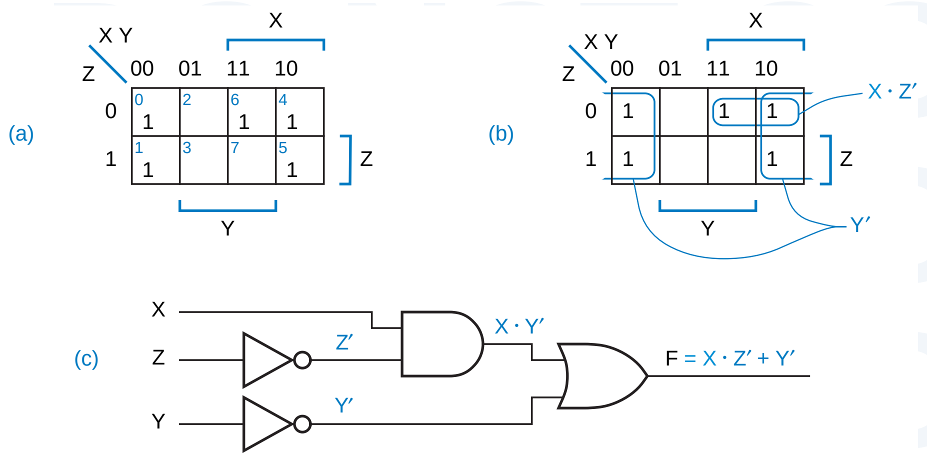


Figure 4-29
F = $\Sigma_{X,Y,Z}(0,1,4,5,6)$:
(a) initial Karnaugh map; (b) Karnaugh map with circled product terms;
(c) AND/OR circuit.

A sum-of-products expression for a function must contain product terms (circled sets of 1-cells) that cover all of the 1s and none of the 0s on the map.

The Karnaugh map for our most recent example, F = $\Sigma_{X,Y,Z}(0, 1, 4, 5, 6)$, is shown in Figure 4-29(a) and (b). We have circled one set of four 1s, corresponding to the product term Y′, and a set of two 1s corresponding to the product term X · Z′. Notice that the second product term has one less literal than the corresponding product term in our algebraic solution (X · Y · Z′). By circling the largest possible set of 1s containing cell 6, we have found a less expensive realization of the logic function, since a 2-input AND gate should cost less than a 3-input one. The fact that two different product terms now cover the same 1-cell (4) does not affect the logic function, since for logical addition 1 + 1 = 1, not 2! The corresponding two-level AND/OR circuit is shown in (c).

As another example, the prime-number detector circuit that we introduced in Figure 4-18 on page 213 can be minimized as shown in Figure 4-30.

At this point, we need some more definitions to clarify what we're doing:

minimal sum

- A *minimal sum* of a logic function F(X_1,…,X_n) is a sum-of-products expression for F such that no sum-of-products expression for F has fewer product terms, and any sum-of-products expression with the same number of product terms has at least as many literals.

That is, the minimal sum has the fewest possible product terms (first-level gates and second-level gate inputs) and, within that constraint, the fewest possible literals (first-level gate inputs). Thus, among our three prime-number detector circuits, only the one in Figure 4-30 on the next page realizes a minimal sum.

The next definition says precisely what the word "imply" means when we talk about logic functions:

imply

- A logic function P(X_1,…,X_n) *implies* a logic function F(X_1,…,X_n) if for every input combination such that P = 1, then F = 1 also.

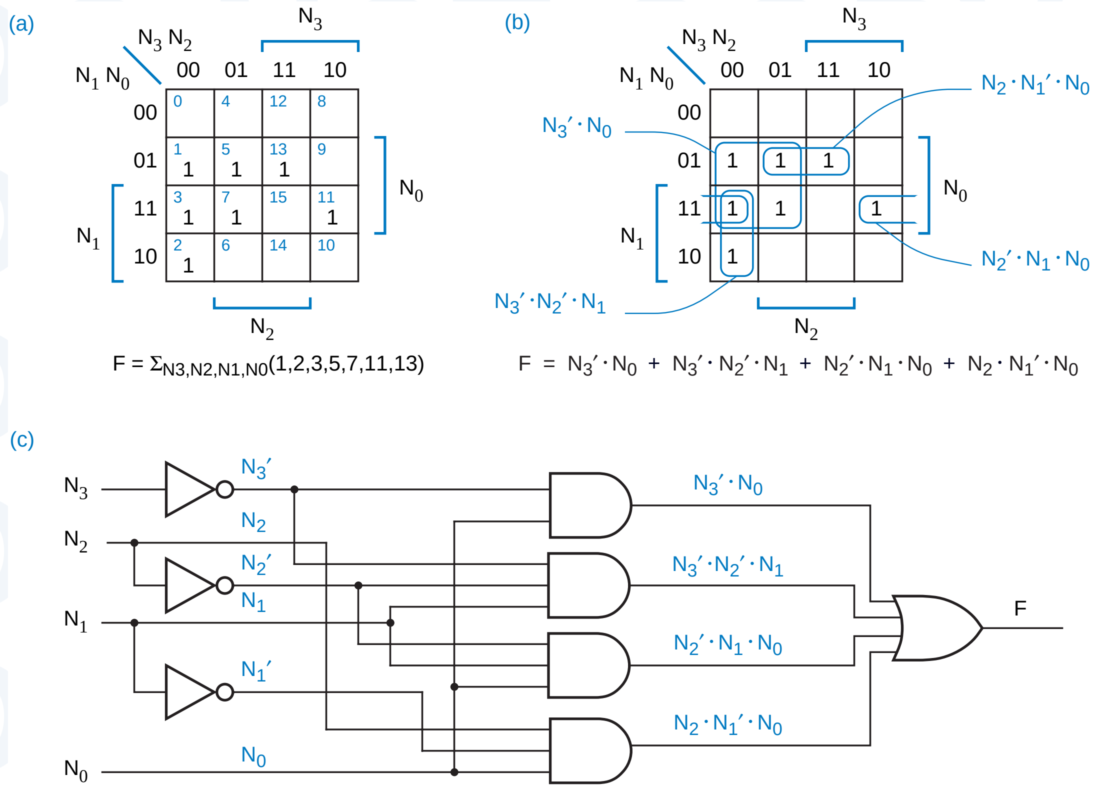


Figure 4-30 Prime-number detector: (a) initial Karnaugh map; (b) circled product terms; (c) minimized circuit.

That is, if P implies F, then F is 1 for every input combination that P is 1, and maybe some more. We may write the shorthand P ⇒ F. We may also say that "F *includes* P," or that "F *covers* P."

includes
covers
prime implicant

- A *prime implicant* of a logic function $F(X_1,\ldots,X_n)$ is a normal product term $P(X_1,\ldots,X_n)$ that implies F, such that if any variable is removed from P, then the resulting product term does not imply F.

In terms of a Karnaugh map, a prime implicant of F is a circled set of 1-cells satisfying our combining rule, such that if we try to make it larger (covering twice as many cells), it covers one or more 0s.

Now comes the most important part, a theorem that limits how much work we must do to find a minimal sum for a logic function:

Prime Implicant Theorem A minimal sum is a sum of prime implicants.

That is, to find a minimal sum, we need not consider any product terms that are not prime implicants. This theorem is easily proved by contradiction. Suppose

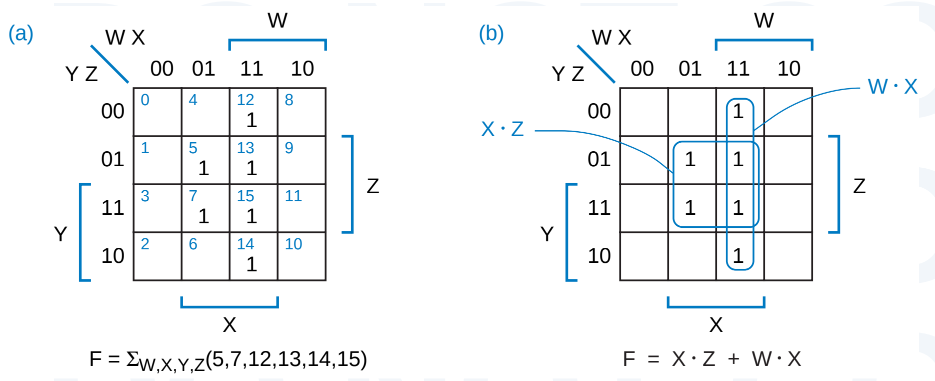


Figure 4-31 $F = \Sigma_{W,X,Y,Z}(5,7,12,13,14,15)$: (a) Karnaugh map; (b) prime implicants.

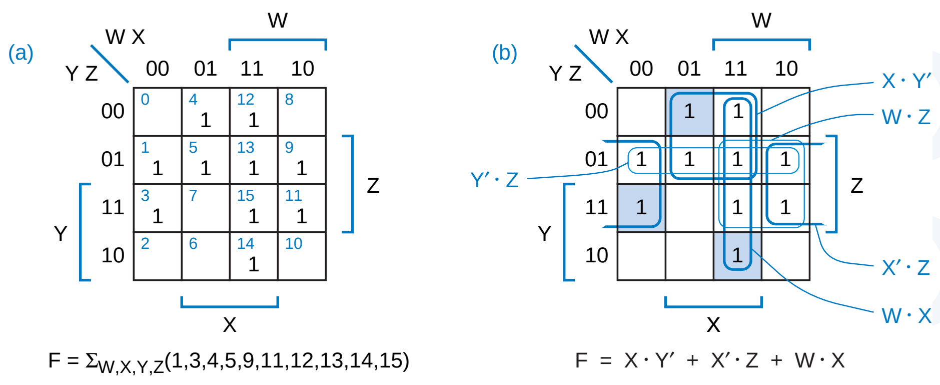
that a product term P in a "minimal" sum is *not* a prime implicant. Then according to the definition of prime implicant, if P is not one, it is possible to remove some literal from P to obtain a new product term P* that still implies F. If we replace P with P* in the presumed "minimal" sum, the resulting sum still equals F but has one fewer literal. Therefore, the presumed "minimal" sum was not minimal after all.

Another minimization example, this time a 4-variable function, is shown in Figure 4-31. There are just two prime implicants, and it's quite obvious that both of them must be included in the minimal sum in order to cover all of the 1-cells on the map. We didn't draw the logic diagram for this example because you should know how to do that yourself by now.

complete sum

The sum of all the prime implicants of a logic function is called the *complete sum*. Although the complete sum is always a legitimate way to realize a logic function, it's not always minimal. For example, consider the logic function shown in Figure 4-32. It has five prime implicants, but the minimal sum includes

Figure 4-32 $F = \Sigma_{W,X,Y,Z}(1,3,4,5,9,11,12,13,14,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells.


(a)
W X
Y Z
W
00 01 11 10
Z
Y
X
F = Σ_{W,X,Y,Z}(1,3,4,5,9,11,12,13,14,15)
(b)
X · Y′
W · Z
Y′ · Z
X′ · Z
W · X
F = X · Y′ + X′ · Z + W · X

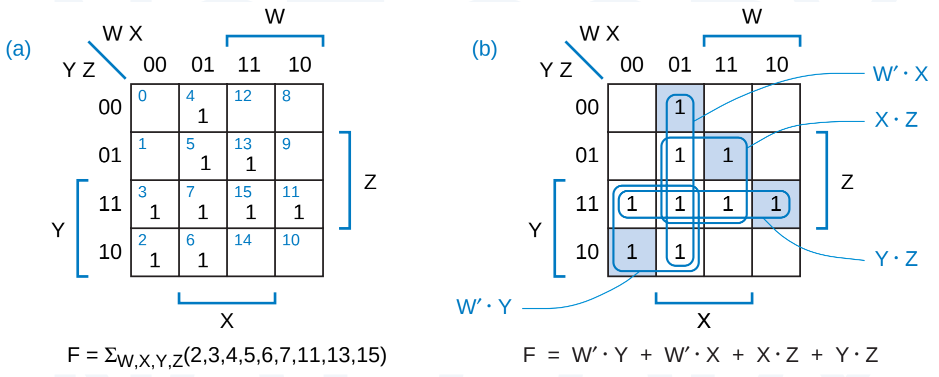


Figure 4-33 F =$\Sigma_{W,X,Y,Z}(2,3,4,5,6,7,11,13,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells.

only three of them. So, how can we systematically determine which prime implicants to include and which to leave out? Two more definitions are needed:

distinguished 1-cell

- A *distinguished 1-cell* of a logic function is an input combination that is covered by only one prime implicant.

essential prime implicant

- An *essential prime implicant* of a logic function is a prime implicant that covers one or more distinguished 1-cells.

Since an essential prime implicant is the *only* prime implicant that covers some 1-cell, it *must* be included in every minimal sum for the logic function. So, the first step in the prime implicant selection process is simple—we identify distinguished 1-cells and the corresponding prime implicants, and include the essential prime implicants in the minimal sum. Then we need only determine how to cover the 1-cells, if any, that are not covered by the essential prime implicants. In the example of Figure 4-32, the three distinguished 1-cells are shaded, and the corresponding essential prime implicants are circled with heavier lines. All of the 1-cells in this example are covered by essential prime implicants, so we need go no further. Likewise, Figure 4-33 shows an example where all of the prime implicants are essential, and so all are included in the minimal sum.

A logic function in which not all the 1-cells are covered by essential prime implicants is shown in Figure 4-34. By removing the essential prime implicants and the 1-cells they cover, we obtain a reduced map with only a single 1-cell and two prime implicants that cover it. The choice in this case is simple—we use the W′ · Z product term because it has fewer inputs and therefore lower cost.

For more complex cases, we need yet another definition:

eclipse

- Given two prime implicants P and Q in a reduced map, P is said to eclipse Q (written P … Q) if P covers at least all the 1-cells covered by Q.

If P costs no more than Q and eclipses Q, then removing Q from consideration cannot prevent us from finding a minimal sum; that is, P is at least as good as Q.

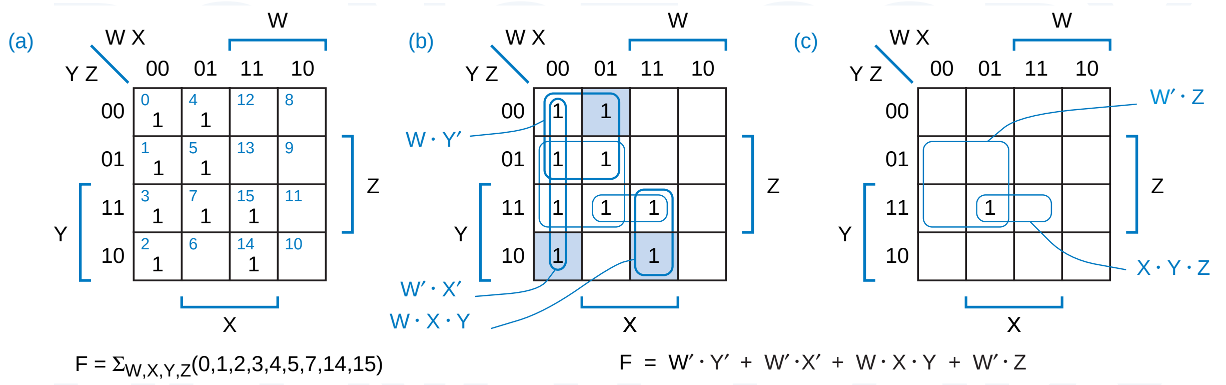

(a)
W X
Y Z
W
00 01 11 10
Z
Y
X
F = $\Sigma_{W,X,Y,Z}$(0,1,2,3,4,5,7,14,15)
(b)
W · Y′
W′ · X′
W · X · Y
(c)
W′ · Z
X · Y · Z
F = W′ · Y′ + W′ · X′ + W · X · Y + W′ · Z

Figure 4-34 $F = \Sigma_{W,X,Y,Z}(0,1,2,3,4,5,7,14,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells; (c) reduced map after removal of essential prime implicants and covered 1-cells.

An example of eclipsing is shown in Figure 4-35. After removing essential prime implicants, we are left with two 1-cells, each of which is covered by two prime implicants. However, X · Y · Z eclipses the other two prime implicants, which therefore may be removed from consideration. The two 1-cells are then covered only by X · Y · Z, which is a *secondary essential prime implicant* that must be included in the minimal sum.

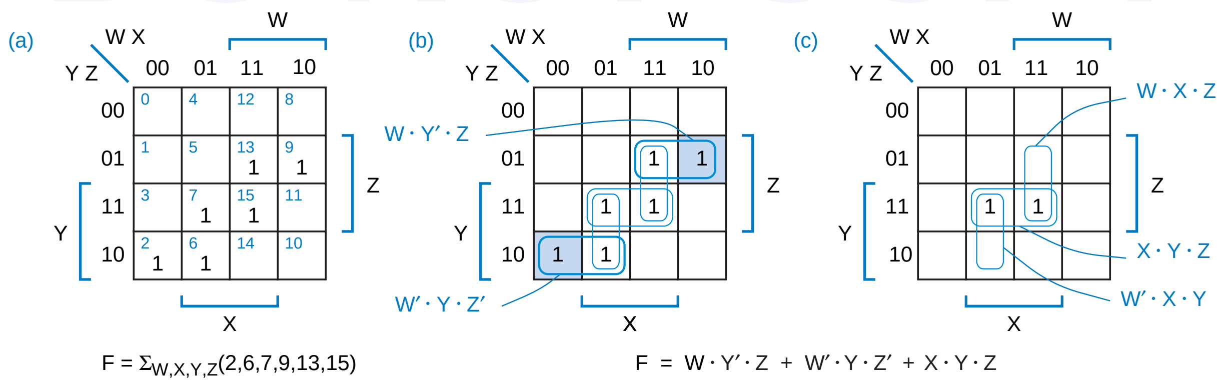
secondary essential prime implicant

Figure 4-36 shows a more difficult case—a logic function with no essential prime implicants. By trial and error we can find two different minimal sums for this function.

We can also approach the problem systematically using the *branching method*. Starting with any 1-cell, we arbitrarily select one of the prime implicants that covers it, and include it as if it were essential. This simplifies the

branching method

Figure 4-35 $F = \Sigma_{W,X,Y,Z}(2,6,7,9,13,15)$: (a) Karnaugh map; (b) prime implicants and distinguished 1-cells; (c) reduced map after removal of essential prime implicants and covered 1-cells.



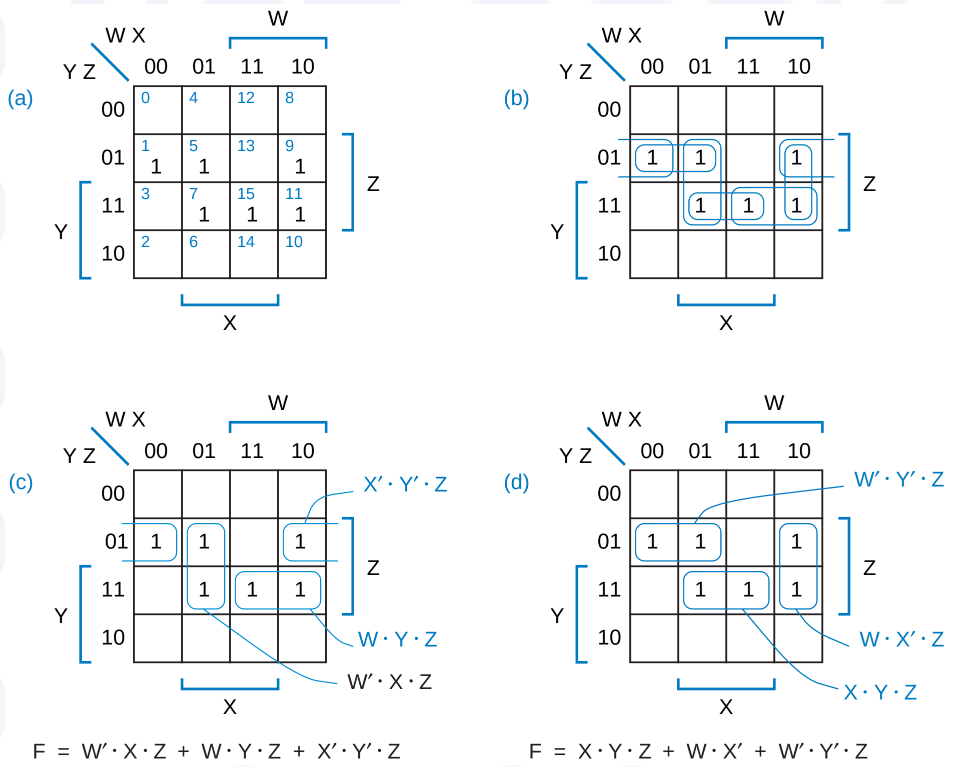


Figure 4-36 $F = \Sigma_{W,X,Y,Z}(1,5,7,9,11,15)$: (a) Karnaugh map; (b) prime implicants; (c) a minimal sum; (d) another minimal sum.

remaining problem, which we can complete in the usual way to find a tentative minimal sum. We repeat this process starting with all other prime implicants that cover the starting 1-cell, generating a different tentative minimal sum from each starting point. We may get stuck along the way and have to apply the branching method recursively. Finally, we examine all of the tentative minimal sums generated in this way and select one that is truly minimal.

4.3.6 Simplifying Products of Sums

Using the principle of duality, we can minimize product-of-sums expressions by looking at the 0s on a Karnaugh map. Each 0 on the map corresponds to a maxterm in the canonical product of the logic function. The entire process in the preceding subsection can be reformulated in a dual way, including the rules for writing sum terms corresponding to circled sets of 0s, in order to find a *minimal product*.

minimal product

Fortunately, once we know how to find minimal sums, there's an easier way to find the minimal product for a given logic function F. The first step is to complement F to obtain F′. Assuming that F is expressed as a minterm list or a truth table, complementing is very easy; the 1s of F′ are just the 0s of F. Next, we find a minimal sum for F′ using the method of the preceding subsection. Finally, we complement the result using the generalized DeMorgan's theorem, which yields a minimal product for (F′)′ = F. (Note that if you simply "add out" the minimal-sum expression for the original function, the resulting product-of-sums expression is not necessarily minimal; for example, see Exercise 4.56.)

In general, to find the lowest-cost two-level realization of a logic function, we have to find both a minimal sum and a minimal product, and compare them. If a minimal sum for a logic function has many terms, then a minimal product for the same function may have few terms. As a trivial example of this tradeoff, consider a 4-input OR function:

$$\begin{aligned} F &= (W) + (X) + (Y) + (Z) \text{ (a sum of four trivial product terms)} \\ &= (W + X + Y + Z) \text{ (a product of one sum term)} \end{aligned}$$

For a nontrivial example, you're invited to find the minimal product for the function that we minimized in Figure 4-33 on page 226; it has just two sum terms.

The opposite situation is also sometimes true, as trivially illustrated by a 4-input AND:

$$\begin{aligned} F &= (W) \cdot (X) \cdot (Y) \cdot (Z) \text{ (a product of four trivial sum terms)} \\ &= (W \cdot X \cdot Y \cdot Z) \text{ (a sum of one product term)} \end{aligned}$$

A nontrivial example with a higher-cost product-of-sums is the function in Figure 4-29 on page 223.

For some logic functions, both minimal forms are equally costly. For example, consider a 3-input "exclusive OR" function; both minimal expressions have four terms, and each term has three literals:

$$\begin{aligned} F &= \Sigma_{X,Y,Z}(1,2,4,7) \\ &= (X' \cdot Y' \cdot Z) + (X' \cdot Y \cdot Z') + (X \cdot Y' \cdot Z') + (X \cdot Y \cdot Z) \\ &= (X + Y + Z) \cdot (X + Y' + Z') \cdot (X' + Y + Z') \cdot (X' + Y' + Z) \end{aligned}$$

Still, in most cases, one form or the other will give better results. Looking at both forms is especially useful in PLD-based designs.

PLD MINIMIZATION

Typical PLDs have an AND-OR array corresponding to a sum-of-products form, so you might think that only the minimal sum-of-products is relevant to a PLD-based design. However, most PLDs also have a programmable inverter/buffer at the output of the AND-OR array, which can either invert or not. Thus, the PLD can utilize the equivalent of the minimal sum by using the AND-OR array to realize the complement of the desired function, and then programming the inverter/buffer to invert. Most logic minimization programs for PLDs automatically find both the minimal sum and the minimal product, and select the one that requires fewer terms.

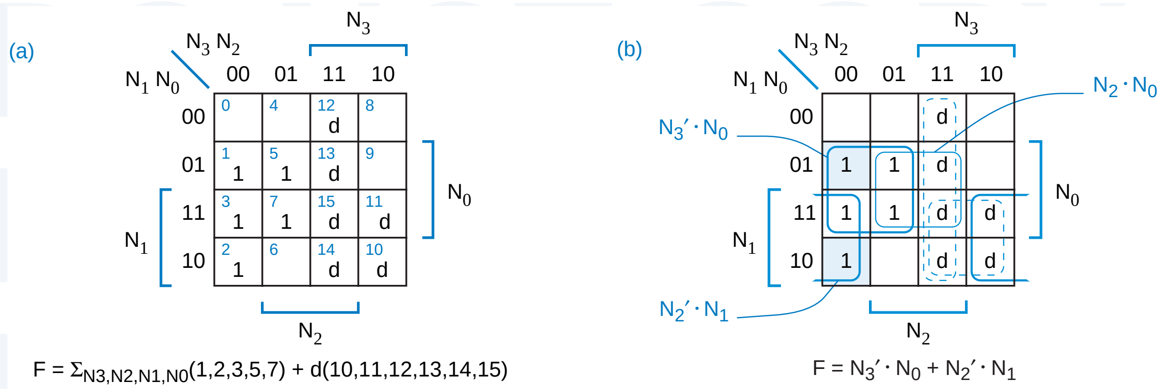


Figure 4-37 Prime BCD-digit detector: (a) initial Karnaugh map; (b) Karnaugh map with prime implicants and distinguished 1-cells.

*4.3.7 "Don't-Care" Input Combinations

don't-care

Sometimes the specification of a combinational circuit is such that its output doesn't matter for certain input combinations, called *don't-cares*. This may be true because the outputs really don't matter when these input combinations occur, or because these input combinations never occur in normal operation. For example, suppose we wanted to build a prime-number detector whose 4-bit input $N = N_3N_2N_1N_0$ is always a BCD digit; then minterms 10–15 should never occur. A prime BCD-digit detector function may therefore be written as follows:

$$F = \Sigma_{N_3,N_2,N_1,N_0}(1,2,3,5,7) + d(10,11,12,13,14,15)$$

d-set

The d(...) list specifies the don't-care input combinations for the function, also known as the *d-set*. Here F must be 1 for input combinations in the on-set (1,2,3,5,7), F can have any values for inputs in the d-set (10,11,12,13,14,15), and F must be 0 for all other input combinations (in the 0-set).

Figure 4-37 shows how to find a minimal sum-of-products realization for the prime BCD-digit detector, including don't-cares. The d's in the map denote the don't-care input combinations. We modify the procedure for circling sets of 1s (prime implicants) as follows:

- Allow d's to be included when circling sets of 1s, to make the sets as large as possible. This reduces the number of variables in the corresponding prime implicants. Two such prime implicants ($N_2 \cdot N_0$ and $N_2' \cdot N_1$) appear in the example.
- Do not circle any sets that contain only d's. Including the corresponding product term in the function would unnecessarily increase its cost. Two such product terms ($N_3 \cdot N_2$ and $N_3 \cdot N_1$) are circled in the example.
- Just a reminder: As usual, do not circle any 0s.

* Throughout this book, optional sections are marked with an asterisk.

The remainder of the procedure is the same. In particular, we look for distinguished 1-cells and *not* distinguished d-cells, and we include only the corresponding essential prime implicants and any others that are needed to cover all the 1s on the map. In Figure 4-37, the two essential prime implicants are sufficient to cover all of the 1s on the map. Two of the d's also happen to be covered, so F will be 1 for don't-care input combinations 10 and 11, and 0 for the other don't-cares.

Some HDLs, including ABEL, provide a means for the designer to specify don't-care inputs, and the logic minimization program takes these into account when computing a minimal sum.

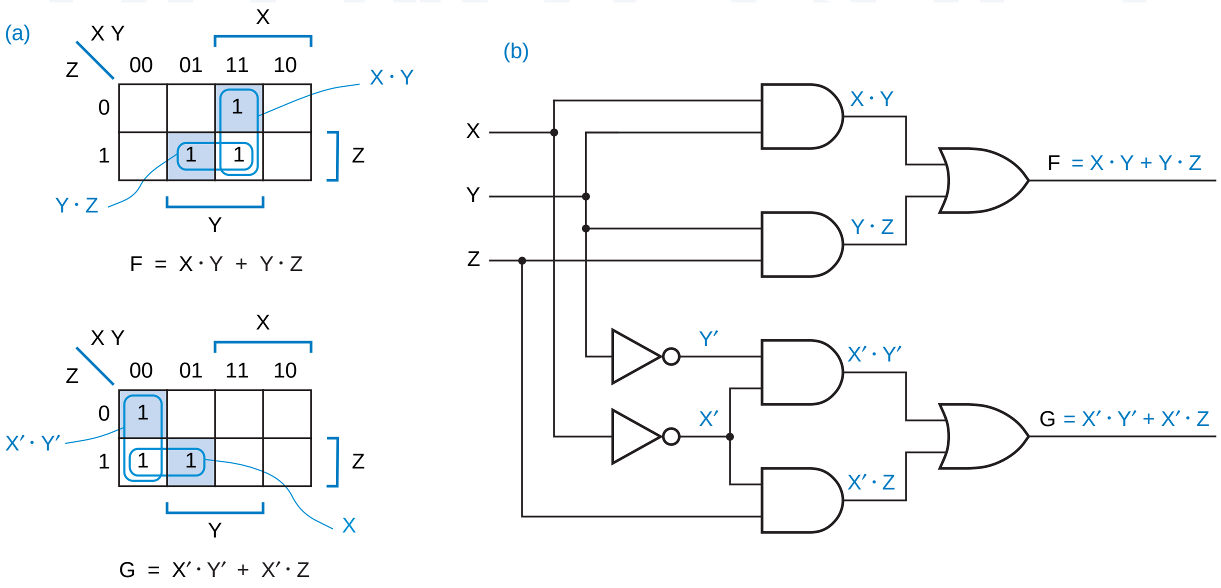
*4.3.8 Multiple-Output Minimization

Most practical combinational logic circuits require more than one output. We can always handle a circuit with *n* outputs as *n* independent single-output design problems. However, in doing so, we may miss some opportunities for optimization. For example, consider the following two logic functions:

$$F = \Sigma_{X,Y,Z}(3,6,7)$$
$$G = \Sigma_{X,Y,Z}(0,1,3)$$

Figure 4-38 shows the design of F and G as two independent single-output functions. However, as shown in Figure 4-39, we can also find a pair of sum-of-products expressions that share a product term, such that the resulting circuit has one fewer gate than our original design.

Figure 4-38 Treating a 2-output design as two independent single-output designs: (a) Karnaugh maps; (b) "minimal" circuit.



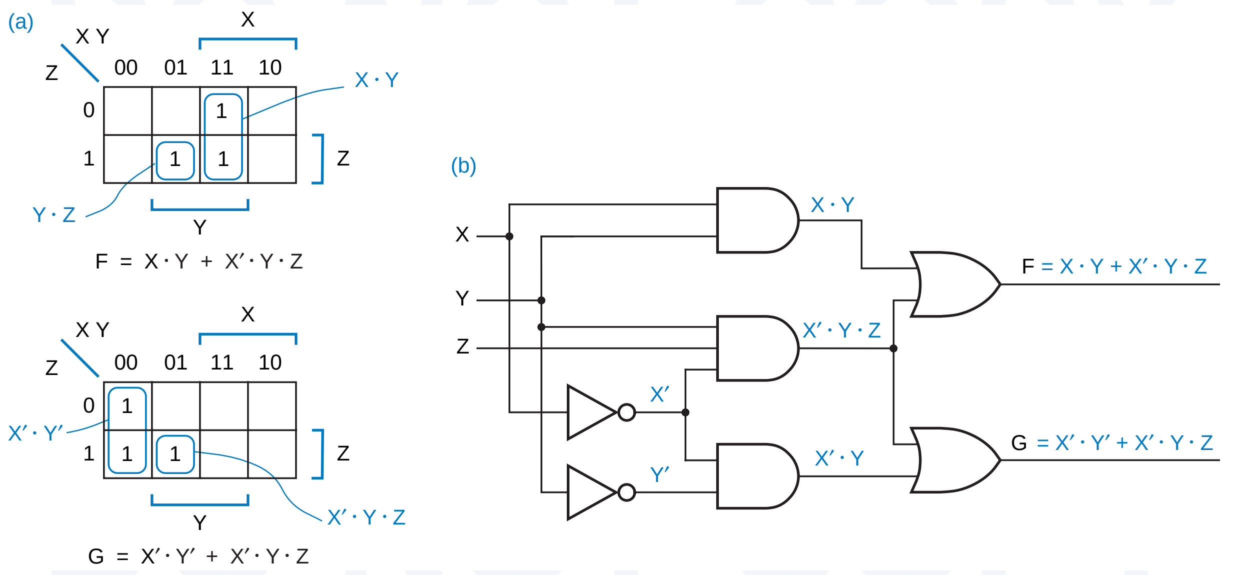


Figure 4-39 Multiple-output minimization for a 2-output circuit: (a) minimized maps including a shared term; (b) minimal multiple-output circuit

When we design multiple-output combinational circuits using discrete gates, as in an ASIC, product-term sharing obviously reduces circuit size and cost. In addition, PLDs contain multiple copies the sum-of-products structure that we've been learning how to minimize, one per output, and some PLDs allow product terms to be shared among multiple outputs. Thus, the ideas introduced in this subsection are used in many logic minimization programs.

You probably could have "eyeballed" the Karnaugh maps for F and G in Figure 4-39, and discovered the minimal solution. However, larger circuits can be minimized only with a formal multiple-output minimization algorithm. We'll outline the ideas in such an algorithm here; details can be found in the References.

The key to successful multiple-output minimization of a set of *n* functions is to consider not only the *n* original single-output functions, but also "product functions." An *m-product function* of a set of *n* functions is the product of *m* of the functions, where $2 \le m \le n$. There are $2^n - n - 1$ such functions. Fortunately, $n = 2$ in our example and there is only one product function, F · G, to consider. The Karnaugh maps for F, G, and F · G are shown in Figure 4-40; in general, the map for an *m*-product function is obtained by ANDing the maps of its *m* components.

m-product function

A *multiple-output prime implicant* of a set of *n* functions is a prime implicant of one of the *n* functions or of one of the product functions. The first step in multiple-output minimization is to find all of the multiple-output

multiple-output prime implicant

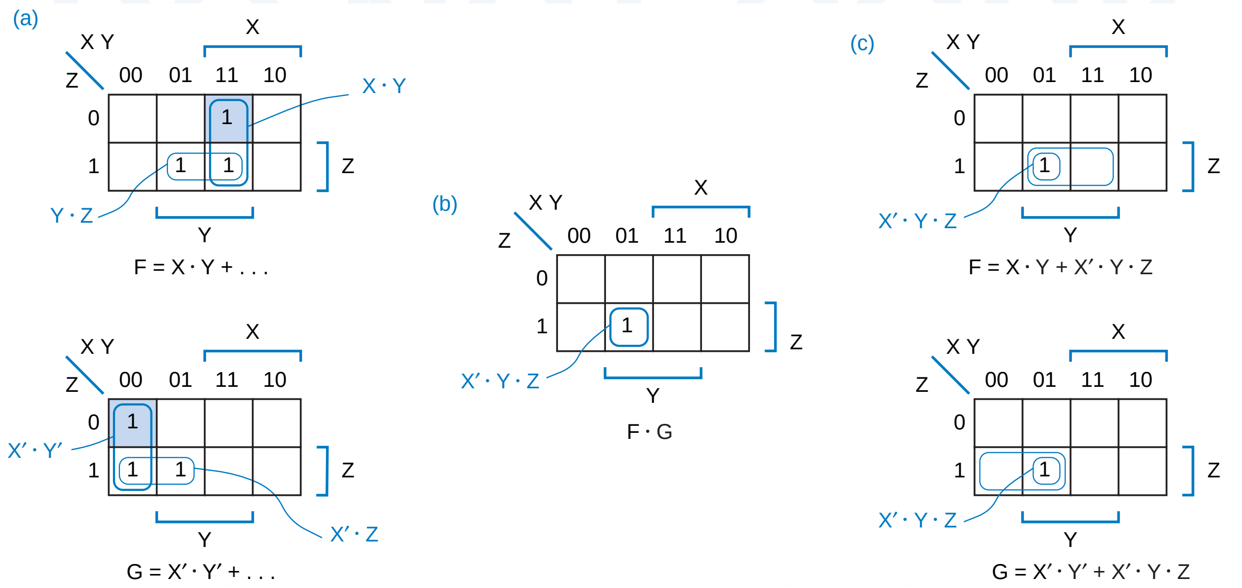


Figure 4-40 Karnaugh maps for a set of two functions: (a) maps for F and G; (b) 2-product map for F · G; (c) reduced maps for F and G after removal of essential prime implicants and covered 1-cells.

prime implicants. Each prime implicant of an *m*-product function is a possible term to include in the corresponding *m* outputs of the circuit. If we were trying to minimize a set of 8 functions, we would have to find the prime implicants for $2^8 - 8 - 1 = 247$ product functions as well as for the 8 given functions. Obviously, multiple-output minimization is not for the faint-hearted!

Once we have found the multiple-output prime implicants, we try to simplify the problem by identifying the essential ones. A *distinguished 1-cell* of a particular single-output function F is a 1-cell that is covered by exactly one prime implicant of F or of the product functions involving F. The distinguished 1-cells in Figure 4-39 are shaded. An *essential prime implicant* of a particular single-output function is one that contains a distinguished 1-cell. As in single-output minimization, the essential prime implicants must be included in a minimum-cost solution. Only the 1-cells that are not covered by essential prime implicants are considered in the remainder of the algorithm.

distinguished 1-cell

essential prime implicant

The final step is to select a minimal set of prime implicants to cover the remaining 1-cells. In this step we must consider all *n* functions simultaneously, including the possibility of sharing; details of this procedure are discussed in the References. In the example of Figure 4-40(c), we see that there exists a single, shared product term that covers the remaining 1-cell in both F and G.

*4.4 Programmed Minimization Methods

Obviously, logic minimization can be a very involved process. In real logic-design applications, you are likely to encounter only two kinds of minimization problems: functions of a few variables that you can "eyeball" using the methods of the previous section, and more complex, multiple-output functions that are hopeless without the use of a minimization program.

Quine-McCluskey algorithm

We know that minimization can be performed visually for functions of a few variables using the Karnaugh-map method. We'll show in this section that the same operations can be performed for functions of an arbitrarily large number of variables (at least in principle) using a tabular method called the *Quine-McCluskey algorithm*. Like all algorithms, the Quine-McCluskey algorithm can be translated into a computer program. And like the map method, the algorithm has two steps: (a) finding all prime implicants of the function, and (b) selecting a minimal set of prime implicants that covers the function.

The Quine-McCluskey algorithm is often described in terms of handwritten tables and manual check-off procedures. However, since no one ever uses these procedures manually, it's more appropriate for us to discuss the algorithm in terms of data structures and functions in a high-level programming language. The goal of this section is to give you an appreciation for computational complexity involved in a large minimization problem. We consider only fully specified, single-output functions; don't-cares and multiple-output functions can be handled by fairly straightforward modifications to the single-output algorithms, as discussed in the References.

*4.4.1 Representation of Product Terms

The starting point for the Quine-McCluskey minimization algorithm is the truth table or, equivalently, the minterm list of a function. If the function is specified differently, it must first be converted into this form. For example, an arbitrary n-variable logic expression can be multiplied out (perhaps using DeMorgan's theorem along the way) to obtain a sum-of-products expression. Once we have a sum-of-products expression, each p-variable product term produces 2^{n-p} minterms in the minterm list.

We showed in Section 4.1.6 that a minterm of an n-variable logic function can be represented by an n-bit integer (the minterm number), where each bit indicates whether the corresponding variable is complemented or uncomplemented. However, a minimization algorithm must also deal with product terms that are not minterms, where some variables do not appear at all. Thus, we must represent three possibilities for each variable in a general product term:

1	Uncomplemented.
0	Complemented.
x	Doesn't appear.

These possibilities are represented by a string of *n* of the above digits in the *cube representation* of a product term. For example, if we are working with product terms of up to eight variables, X7, X6, …, X1, X0, we can write the following product terms and their cube representations:

cube representation

$$\begin{aligned} X7' \cdot X6 \cdot X5 \cdot X4' \cdot X3 \cdot X2 \cdot X1 \cdot X0' &\equiv 01101110 \\ X3 \cdot X2 \cdot X1 \cdot X0' &\equiv \text{xxxx1110} \\ X7 \cdot X5' \cdot X4 \cdot X3 \cdot X2' \cdot X1 &\equiv \text{1x01101x} \\ X6 &\equiv \text{x1xxxxxx} \end{aligned}$$

Notice that for convenience, we named the variables just like the bit positions in *n*-bit binary integers.

In terms of the *n*-cube and *m*-subcube nomenclature of Section 2.14, the string 1x01101x represents a 2-subcube of an 8-cube, and the string 01101110 represents a 0-subcube of an 8-cube. However, in the minimization literature, the maximum dimension *n* of a cube or subcube is usually implicit, and an *m*-subcube is simply called an "*m*-cube" or a "cube" for short; we'll follow this practice in this section.

To represent a product term in a computer program, we can use a data structure with *n* elements, each of which has three possible values. In C, we might make the following declarations:

```
typedef enum {complemented, uncomplemented, doesntappear} TRIT;
typedef TRIT[16] CUBE;  /* Represents a single product
                           term with up to 16 variables */
```

However, these declarations do not lead to a particularly efficient internal representation of cubes. As we'll see, cubes are easier to manipulate using conventional computer instructions if an *n*-variable product term is represented by two n-bit computer words, as suggested by the following declarations:

```
#define MAX_VARS 16     /* Max # of variables in a product term */
typedef unsigned short WORD;   /* Use 16-bit words */
struct cube {
  WORD t;  /* Bits 1 for uncomplemented variables. */
  WORD f;  /* Bits 1 for complemented variables.   */
};
typedef struct cube CUBE;
CUBE P1, P2, P3;    /* Allocate three cubes for use by program. */
```

Here, a `WORD` is a 16-bit integer, and a 16-variable product term is represented by a record with two `WORD`s, as shown in Figure 4-41(a). The first word in a `CUBE` has a 1 for each variable in the product term that appears uncomplemented (or "true," `t`), and the second has a 1 for each variable that appears complemented (or "false," `f`). If a particular bit position has 0s in both `WORD`s, then the corresponding variable does not appear, while the case of a particular bit position

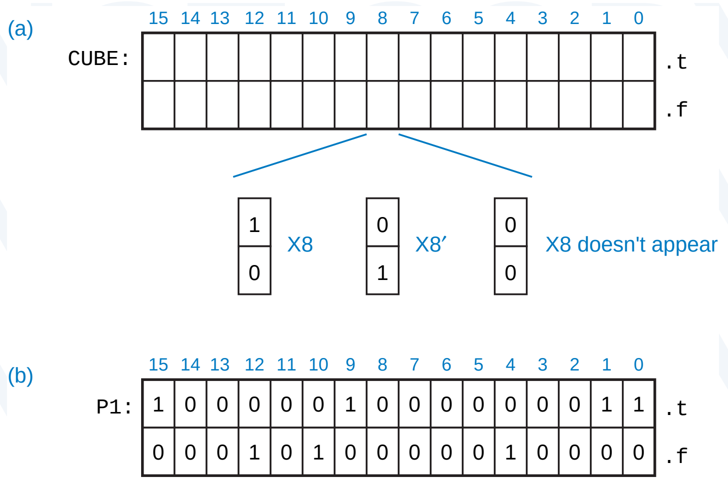


Figure 4-41 Internal representation of 16-variable product terms in a Pascal program: (a) general format; (b) P1 = X15·X12′·X10′·X9·X4′·X1·X0

having 1s in both WORDs is not used. Thus, the program variable P1 in (b) represents the product term P1 = X15 · X12′ · X10′ · X9 · X4′ · X1 · X0. If we wished to represent a logic function F of up to 16 variables, containing up to 100 product terms, we could declare an array of 100 CUBEs:

```
CUBE F[100];        /* Storage for a logic function
                       with up to 100 product terms. */
```

Using the foregoing cube representation, it is possible to write short, efficient C functions that manipulate product terms in useful ways. Table 4-8 shows several such functions. Corresponding to two of the functions, Figure 4-42 depicts how two cubes can be compared and combined if possible

Figure 4-42 Cube manipulations: (a) determining whether two cubes are combinable using theorem T10, term · X + term · X′ = term; (b) combining cubes using theorem T10.

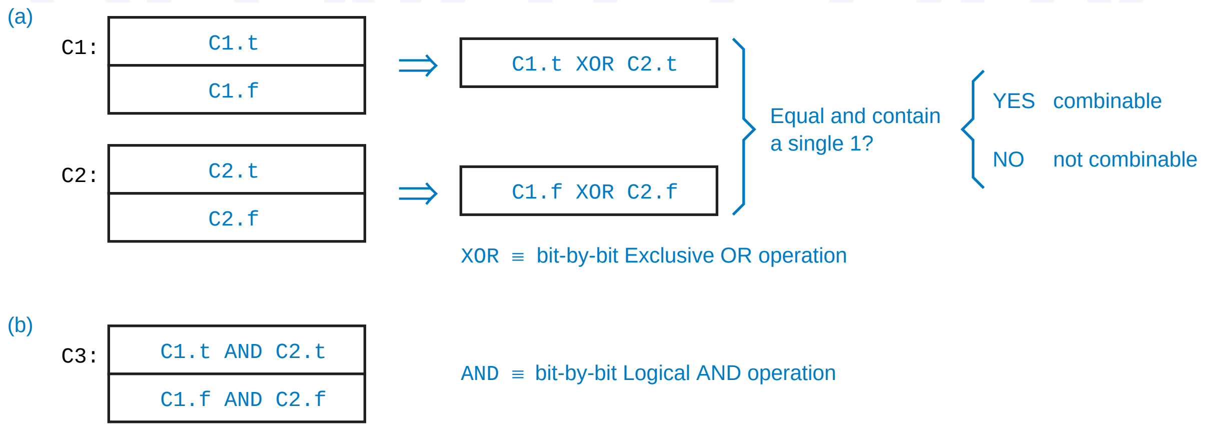


Table 4-8 Cube comparing and combining functions used in minimization program.

```
int EqualCubes(CUBE C1, CUBE C2)       /* Returns true if C1 and C2 are identical.  */
{
  return ( (C1.t == C2.t) && (C1.f == C2.f) );
}

int Oneone(WORD w)            /* Returns true if w has exactly one 1 bit.           */
{                             /* Optimizing the speed of this routine is critical   */
  int ones, b;                /*   and is left as an exercise for the hacker.       */
  ones = 0;
  for (b=0; b<MAX_VARS; b++) {
    if (w & 1) ones++;
    w = w>>1;
  }
  return((ones==1));
}

int Combinable(CUBE C1, CUBE C2)
{                          /* Returns true if C1 and C2 differ in only one variable, */
  WORD twordt, twordf;     /* which appears true in one and false in the other.      */

  twordt = C1.t ^ C2.t;
  twordf = C1.f ^ C2.f;
  return( (twordt==twordf) && Oneone(twordt) );
}

void Combine(CUBE C1, CUBE C2, CUBE *C3)
                         /* Combines C1 and C2 using theorem T10, and stores the    */
{                        /*   result in C3.  Assumes Combinable(C1,C2) is true.     */
  C3->t = C1.t & C2.t;
  C3->f = C1.f & C2.f;
}
```

using theorem T10, term $\cdot$ X + term $\cdot$ X′ = term. This theorem says that two product terms can be combined if they differ in only one variable that appears complemented in one term and uncomplemented in the other. Combining two *m*-cubes yields an $(m+1)$-cube. Using cube representation, we can apply the combining theorem to a few examples:

$$\begin{aligned} 010 + 000 &= 0x0 \\ 00111001 + 00111000 &= 0011100x \\ 101xx0x0 + 101xx1x0 &= 101xxxx0 \\ x111xx00110x000x + x111xx00010x000x &= x111xx00x10x000x \end{aligned}$$

*4.4.2 Finding Prime Implicants by Combining Product Terms

The first step in the Quine-McCluskey algorithm is to determine all of the prime implicants of the logic function. With a Karnaugh map, we do this visually by identifying "largest possible rectangular sets of 1s." In the algorithm, this is done by systematic, repeated application of theorem T10 to combine minterms, then 1-cubes, 2-cubes, and so on, creating the largest possible cubes (smallest possible product terms) that cover only 1s of the function.

The C program in Table 4-9 applies the algorithm to functions with up to 16 variables. It uses 2-dimensional arrays, `cubes[m][j]` and `covered[m][j]`, to keep track of `MAX_VARS` *m*-cubes. The 0-cubes (minterms) are supplied by the user. Starting with the 0-cubes, the program examines every pair of cubes at each level and combines them when possible into cubes at the next level. Cubes that are combined into a next-level cube are marked as "covered"; cubes that are not covered are prime implicants.

Even though the program in Table 4-9 is short, an experienced programmer could become very pessimistic just looking at its structure. The inner `for` loop is nested four levels deep, and the number of times it might be executed is on the order of $\texttt{MAX\_VARS} \cdot \texttt{MAX\_CUBES}^3$. That's right, that's an exponent, not a footnote! We picked the value `maxCubes` = 1000 somewhat arbitrarily (in fact, too optimistically for many functions), but if you believe this number, then the inner loop can be executed *billions and billions* of times.

The maximum number of minterms of an *n*-variable function is 2^n, of course, and so by all rights the program in Table 4-9 should declare `maxCubes` to be 2^{16}, at least to handle the maximum possible number of 0-cubes. Such a declaration would not be overly pessimistic. If an *n*-variable function has a product term equal to a single variable, then 2^{n-1} minterms are in fact needed to cover that product term.

For larger cubes, the situation is actually worse. The number of possible *m*-subcubes of an *n*-cube is $\binom{n}{m} \times 2^{n-m}$, where the binomial coefficient $\binom{n}{m}$ is the number of ways to choose *m* variables to be x's, and 2^{n-m} is the number of ways to assign 0s and 1s to the remaining variables. For 16-variable functions, the worst case occurs with $m = 5$; there are 8,945,664 possible 5-subcubes of a 16-cube. The total number of distinct *m*-subcubes of an *n*-cube, over all values of *m*, is 3^n. So a general minimization program might require a *lot* more memory than we've allocated in Table 4-9.

There are a few things that we can do to optimize the storage space and execution time required in Table 4-9 (see Exercises 4.72–4.75), but they are piddling compared to the overwhelming combinatorial complexity of the problem. Thus, even with today's fast computers and huge memories, direct application of the Quine-McCluskey algorithm for generating prime implicants is generally limited to functions with only a few variables (fewer than 15–20).

Table 4-9 A C program that finds prime implicants using the Quine-McCluskey algorithm.

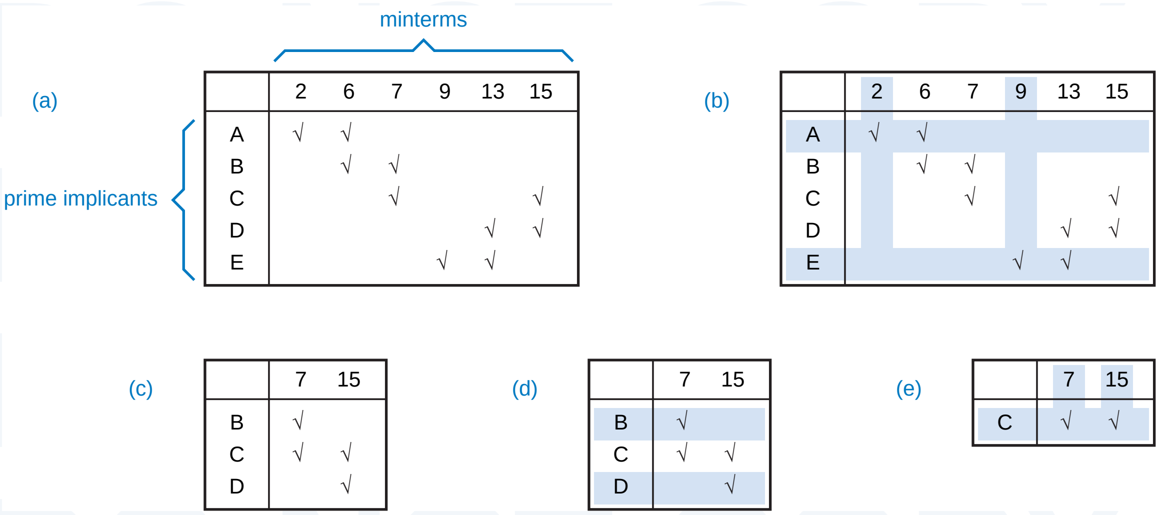
```
#define TRUE    1
#define FALSE   0
#define MAX_CUBES 50

void main()
{
  CUBE cubes[MAX_VARS+1][MAX_CUBES];
  int covered[MAX_VARS+1][MAX_CUBES];
  int numCubes[MAX_VARS+1];
  int m;          /* Value of m in an m-cube, i.e., ‘‘level m.’’ */
  int j, k, p;    /* Indices into the cubes or covered array.    */
  CUBE tempCube;
  int found;

  /* Initialize number of m-cubes at each level m. */
  for (m=0; m<MAX_VARS+1; m++) numCubes[m] = 0;

  /* Read a list of minterms (0-cubes) supplied by the user, storing them    */
  /* in the cubes[0,j] subarray, setting covered[0,j] to false for each      */
  /* minterm, and setting numCubes[0] to the total number of minterms read.  */
ReadMinterms;

  for (m=0; m<MAX_VARS; m++)            /* Do for all levels except the last */
    for (j=0; j<numCubes[m]; j++)       /* Do for all cubes at this level    */
      for (k=j+1; k<numCubes[m]; k++)   /* Do for other cubes at this level  */
        if (Combinable(cubes[m][j], cubes[m][k])) {
          /* Mark the cubes as covered. */
          covered[m][j] = TRUE;  covered[m][k] = TRUE;
          /* Combine into an (m+1)-cube, store in tempCube. */
          Combine(cubes[m][j], cubes[m][k], &tempCube);
          found = FALSE;  /* See if we’ve generated this one before. */
          for (p=0; p<numCubes[m+1]; p++)
            if (EqualCubes(cubes[m+1][p],tempCube)) found = TRUE;
          if (!found) {  /* Add the new cube to the next level. */
            numCubes[m+1] = numCubes[m+1] + 1;
            cubes[m+1][numCubes[m+1]-1] = tempCube;
            covered[m+1][numCubes[m+1]-1] = FALSE;
          }
        }
  for (m=0; m<MAX_VARS; m++)       /* Do for all levels               */
    for (j=0; j<numCubes[m]; j++)  /* Do for all cubes at this level */
      /* Print uncovered cubes -- these are the prime implicants. */
      if (!covered[m][j]) PrintCube(cubes[m][j]);
}
```


minterms
prime implicants
(a)
(b)
(c)
(d)
(e)

(a)

	2	6	7	9	13	15
A	√	√				
B		√	√			
C			√			√
D					√	√
E				√	√	

(b)

	2	6	7	9	13	15
A	√	√				
B		√	√			
C			√			√
D					√	√
E				√	√	

(c)

	7	15
B	√	
C	√	√
D		√

(d)

	7	15
B	√	
C	√	√
D		√

(e)

	7	15
C	√	√

Figure 4-43 Prime-implicant tables: (a) original table; (b) showing distinguished 1-cells and essential prime implicants; (c) after removal of essential prime implicants; (d) showing eclipsed rows; (e) after removal of eclipsed rows, showing secondary essential prime implicant.

*4.4.3 Finding a Minimal Cover Using a Prime-Implicant Table

The second step in minimizing a combinational logic function, once we have a list of all its prime implicants, is to select a minimal subset of them to cover all the 1s of the function. The Quine-McCluskey algorithm uses a two-dimensional array called a *prime-implicant table* to do this. Figure 4-43(a) shows a small but representative prime-implicant table, corresponding to the Karnaugh-map minimization problem of Figure 4-35. There is one column for each minterm of the function, and one row for each prime implicant. Each entry is a bit that is 1 if and only if the prime implicant for that row covers the minterm for that column (shown in the figure as a check).

prime-implicant table

The steps for selecting prime implicants with the table are analogous to the steps that we used in Section 4.3.5 with Karnaugh maps:

1. Identify distinguished 1-cells. These are easily identified in the table as columns with a single 1, as shown in Figure 4-43(b).
2. Include all essential prime implicants in the minimal sum. A row that contains a check in one or more distinguished-1-cell columns corresponds to an essential prime implicant.
3. Remove from consideration the essential prime implicants and the 1-cells (minterms) that they cover. In the table, this is done by deleting the corresponding rows and columns, marked in color in Figure 4-43(b). If any rows

have no checks remaining, they are also deleted; the corresponding prime implicants are *redundant*, that is, completely covered by essential prime implicants. This step leaves the reduced table shown in (c).

redundant prime implicant

4. Remove from consideration any prime implicants that are "eclipsed" by others with equal or lesser cost. In the table, this is done by deleting any rows whose checked columns are a proper subset of another row's, and deleting all but one of a set of rows with identical checked columns. This is shown in color in (d), and leads to the further reduced table in (e).

   When a function is realized in a PLD, all of its prime implicants may be considered to have equal cost, because all of the AND gates in a PLD have all of the inputs available. Otherwise, the prime implicants must be sorted and selected according to the number of AND-gate inputs.

5. Identify distinguished 1-cells and include all secondary essential prime implicants in the minimal sum. As before, any row that contains a check in one or more distinguished-1-cell columns corresponds to a secondary essential prime implicant.

6. If all remaining columns are covered by the secondary essential prime implicants, as in (e), we're done. Otherwise, if any secondary essential prime implicants were found in the previous step, we go back to step 3 and iterate. Otherwise, the branching method must be used, as described in Section 4.3.5. This involves picking rows one at a time, treating them as if they were essential, and recursing (and cursing) on steps 3–6.

Although a prime-implicant table allows a fairly straightforward prime-implicant selection algorithm, the data structure required in a corresponding computer program is huge, since it requires on the order of $p \cdot 2^n$ bits, where p is the number of prime implicants and n is the number of input bits (assuming that the given function produces a 1 output for most input combinations). Worse, executing the steps that we so blithely described in a few sentences above requires a huge amount of computation.

*4.4.4 Other Minimization Methods

Although the previous subsections form an introduction to logic minimization algorithms, the methods they describe are by no means the latest and greatest. Spurred on by the ever increasing density of VLSI chips, many researchers have discovered more effective ways to minimize combinational logic functions. Their results fall roughly into three categories:

1. *Computational improvements.* Improved algorithms typically use clever data structures or rearrange the order of the steps to reduce the memory requirements and execution time of the classical algorithms.
2. *Heuristic methods.* Some minimization problems are just too big to be solved using an "exact" algorithm. These problems can be attacked using

shortcuts and well-educated guesses to reduce memory size and execution time to a fraction of what an “exact” algorithm would require. However, rather than finding a provably minimal expression for a logic function, heuristic methods attempt to find an “almost minimal” one.

Even for problems that can be solved by an “exact” method, a heuristic method typically finds a good solution ten times faster. The most successful heuristic program, Espresso-II, does in fact produce minimal or near-minimal results for the majority of problems (within one or two product terms), including problems with dozens of inputs and hundreds of product terms.

3. *Looking at things differently.* As we mentioned earlier, multiple-output minimization can be handled by straightforward, fairly mechanical modifications to single-output minimization methods. However, by looking at multiple-output minimization as a problem in multivalued (nonbinary) logic, the designers of the Espresso-MV algorithm were able to make substantial performance improvements over Espresso-II.

More information on these methods can be found in the References.

*4.5 Timing Hazards

steady-state behavior

The analysis methods that we developed in Section 4.2 ignore circuit delay and predict only the *steady-state behavior* of combinational logic circuits. That is, they predict a circuit’s output as a function of its inputs under the assumption that the inputs have been stable for a long time, relative to the delays in the circuit’s electronics. However, we showed in Section 3.6 that the actual delay from an input change to the corresponding output change in a real logic circuit is nonzero and depends on many factors.

transient behavior

glitch

hazard

Because of circuit delays, the *transient behavior* of a logic circuit may differ from what is predicted by a steady-state analysis. In particular, a circuit’s output may produce a short pulse, often called a *glitch*, at a time when steady-state analysis predicts that the output should not change. A *hazard* is said to exist when a circuit has the possibility of producing such a glitch. Whether or not the glitch actually occurs depends on the exact delays and other electrical characteristics of the circuit. Since such parameters are difficult to control in production circuits, a logic designer must be prepared to eliminate hazards (the *possibility* of a glitch) even though a glitch may occur only under a worst-case combination of logical and electrical conditions.

*4.5.1 Static Hazards

static-1 hazard

A *static-1 hazard* is the possibility of a circuit’s output producing a 0 glitch when we would expect the output to remain at a nice steady 1 based on a static analysis of the circuit function. A formal definition is given as follows.

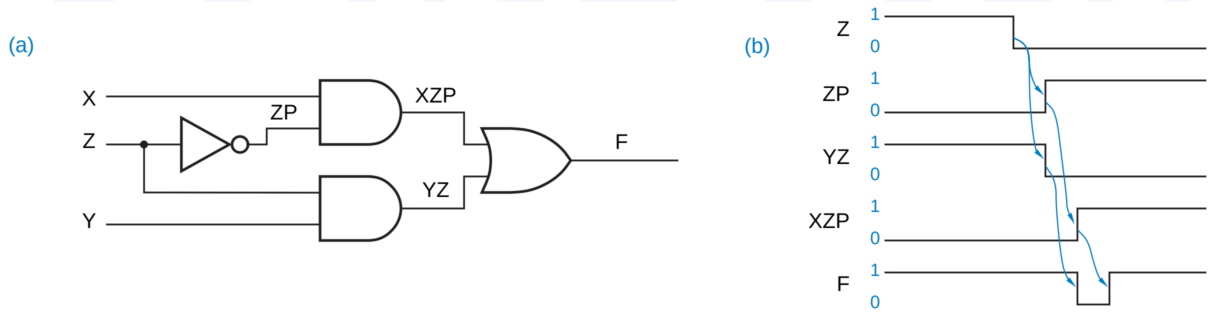


Figure 4-44 Circuit with a static-1 hazard: (a) logic diagram; (b) timing diagram.

Definition: A static-1 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 1 output; such that it is possible for a momentary 0 output to occur during a transition in the differing input variable.

For example, consider the logic circuit in Figure 4-44(a). Suppose that X and Y are both 1, and Z is changing from 1 to 0. Then (b) shows the timing diagram assuming that the propagation delay through each gate or inverter is one unit time. Even though “static” analysis predicts that the output is 1 for both input combinations X,Y,Z = 111 and X,Y,Z = 110, the timing diagram shows that F goes to 0 for one unit time during a 1-0 transition on Z, because of the delay in the inverter that generates Z′.

A *static-0 hazard* is the possibility of a 1 glitch when we expect the circuit to have a steady 0 output:

static-0 hazard

Definition: A static-0 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 0 output; such that it is possible for a momentary 1 output to occur during a transition in the differing input variable.

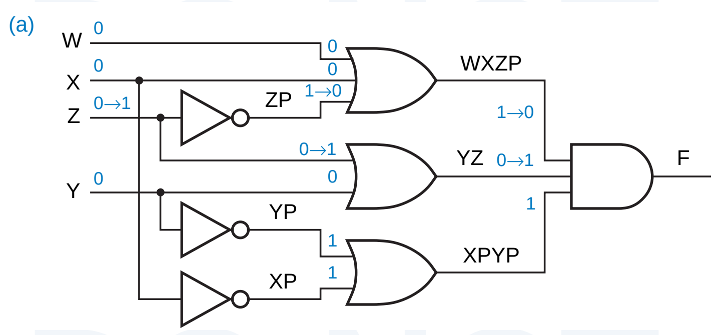
Since a static-0 hazard is just the dual of a static-1 hazard, an OR-AND circuit that is the dual of Figure 4-44(a) would have a static-0 hazard.

An OR-AND circuit with four static-0 hazards is shown in Figure 4-45(a). One of the hazards occurs when W,X,Y = 000 and Z is changed, as shown in (b). You should be able to find the other three hazards and eliminate all of them after studying the next subsection.

*4.5.2 Finding Static Hazards Using Maps

A Karnaugh map can be used to detect static hazards in a two-level sum-of-products or product-of-sums circuit. The existence or nonexistence of static hazards depends on the circuit design for a logic function.

A properly designed two-level sum-of-products (AND-OR) circuit has no static-0 hazards. A static-0 hazard would exist in such a circuit only if both a



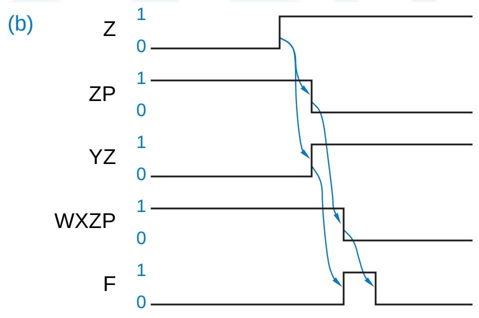


Figure 4-45 Circuit with static-0 hazards: (a) logic diagram; (b) timing diagra

variable and its complement were connected to the same AND gate, which would be silly. However, the circuit *may* have static-1 hazards. Their existence can be predicted from a Karnaugh map where the product terms corresponding to the AND gates in the circuit are circled.

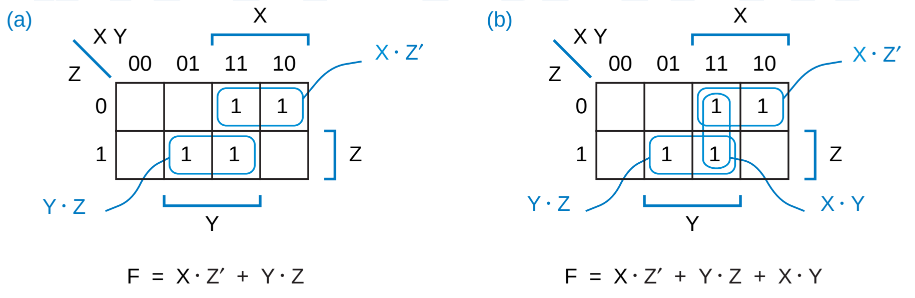
Figure 4-46(a) shows the Karnaugh map for the circuit of Figure 4-44. It is clear from the map that there is no single product term that covers both input combinations X,Y,Z = 111 and X,Y,Z = 110. Thus, intuitively, it is possible for the output to "glitch" momentarily to 0 if the AND gate output that covers one of the combinations goes to 0 before the AND gate output covering the other input combination goes to 1. The way to eliminate the hazard is also quite apparent: Simply include an extra product term (AND gate) to cover the hazardous input pair, as shown in Figure 4-46(b). The extra product term, it turns out, is the *consensus* of the two original terms; in general, we must add consensus terms to eliminate hazards. The corresponding hazard-free circuit is shown in Figure 4-47.

consensus

Another example is shown in Figure 4-48. In this example, three product terms must be added to eliminate the static-1 hazards.

A properly designed two-level product-of-sums (OR-AND) circuit has no static-1 hazards. It *may* have static-0 hazards, however. These hazards can be detected and eliminated by studying the adjacent 0s in the Karnaugh map, in a manner dual to the foregoing.

Figure 4-46
Karnaugh map for the circuit of Figure 4-44:
(a) as originally designed;
(b) with static-1 hazard eliminated.



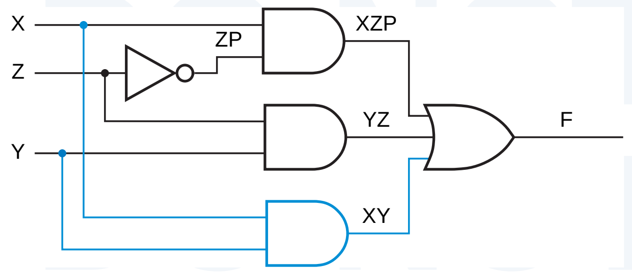


Figure 4-47
Circuit with static-1 hazard eliminated.

*4.5.3 Dynamic Hazards

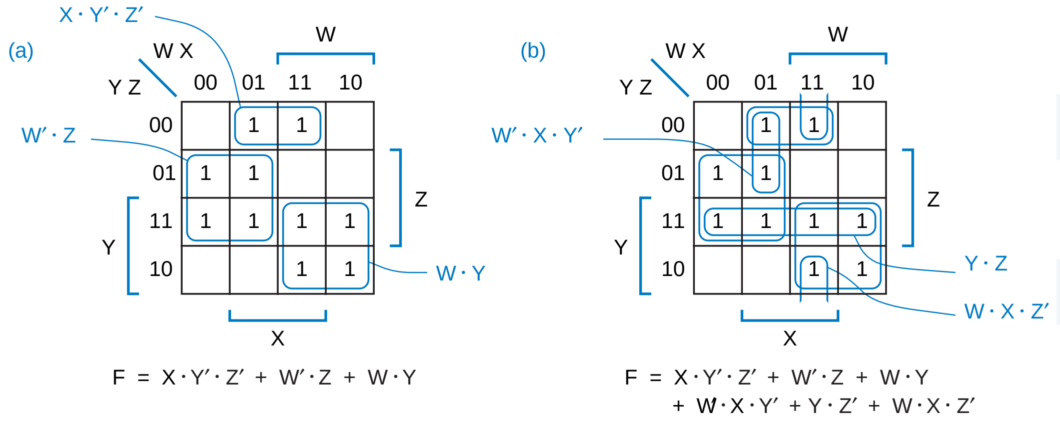
A *dynamic hazard* is the possibility of an output changing more than once as the result of a single input transition. Multiple output transitions can occur if there are multiple paths with different delays from the changing input to the changing output.

dynamic hazard

For example, consider the circuit in Figure 4-49; it has three different paths from input X to the output F. One of the paths goes through a slow OR gate, and another goes through an OR gate that is even slower. If the input to the circuit is W,X,Y,Z = 0,0,0,1, then the output will be 1, as shown. Now suppose we change the X input to 1. Assuming that all of the gates except the two marked “slow” and “slower” are very fast, the transitions shown in black occur next, and the output goes to 0. Eventually, the output of the “slow” OR gate changes, creating the transitions shown in nonitalic color, and the output goes to 1. Finally, the output of the “slower” OR gate changes, creating the transitions shown in italic color, and the output goes to its final state of 0.

Dynamic hazards do not occur in a properly designed two-level AND-OR or OR-AND circuit, that is, one in which no variable and its complement are con-

Figure 4-48 Karnaugh map for another sum-of-products circuit: (a) as originally designed; (b) with extra product terms to cover static-1 hazards.


(a)
X · Y′ · Z′
W′ · Z
W · Y
W X
Y Z
W
00 01 11 10
00 01 11 10
Z
Y
X
F = X · Y′ · Z′ + W′ · Z + W · Y
(b)
W′ · X · Y′
Y · Z
W · X · Z′
F = X · Y′ · Z′ + W′ · Z + W · Y + W′ · X · Y′ + Y · Z + W · X · Z′

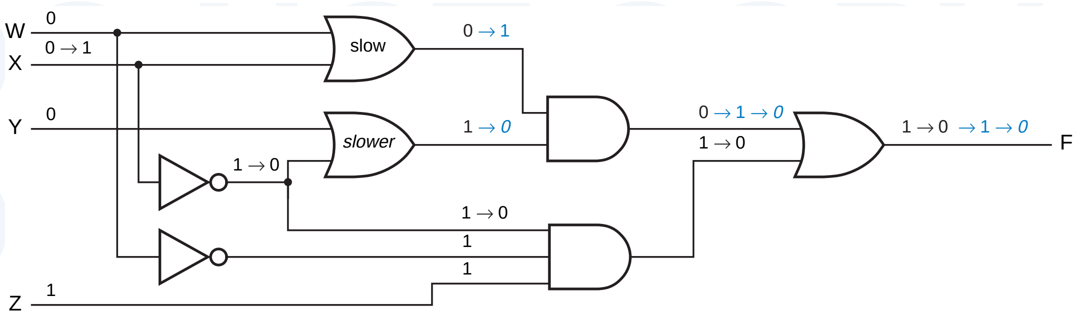


Figure 4-49 Circuit with a dynamic hazard.

nected to the same first-level gate. In multilevel circuits, dynamic hazards can be discovered using a method described in the References.

*4.5.4 Designing Hazard-Free Circuits

There are only a few situations, such as the design of feedback sequential circuits, that require hazard-free combinational circuits. Techniques for finding hazards in arbitrary circuits, described in the References, are rather difficult to use. So, when you require a hazard-free design, it's best to use a circuit structure that is easy to analyze.

In particular, we have indicated that a properly designed two-level AND-OR circuit has no static-0 or dynamic hazards. Static-1 hazards may exist in such a circuit, but they can be found and eliminated using the map method described earlier. If cost is not a problem, then a brute-force method of obtaining a hazard-free realization is to use the complete sum—the sum of all of the prime implicants of the logic function (see Exercise 4.79). In a dual manner, a hazard-free two-level OR-AND circuit can be designed for any logic function. Finally, note that everything we've said about AND-OR circuits naturally applies to the corresponding NAND-NAND designs, and for OR-AND applies to NOR-NOR.

MOST HAZARDS ARE NOT HAZARDOUS!

Any combinational circuit can be analyzed for the presence of hazards. However, a well-designed, *synchronous* digital system is structured so that hazard analysis is not needed for most of its circuits. In a synchronous system, all of the inputs to a combinational circuit are changed at a particular time, and the outputs are not "looked at" until they have had time to settle to a steady-state value. Hazard analysis and elimination are typically needed only in the design of asynchronous sequential circuits, such as the feedback sequential circuits discussed in \secref{fdbkseq}. You'll rarely have reason to design such a circuit, but if you do, an understanding of hazards will be absolutely essential for a reliable result.

4.6 The ABEL Hardware Design Language

ABEL is a hardware design language (HDL) that was invented to allow designers to specify logic functions for realization in PLDs. An ABEL program is a text file containing several elements:

- Documentation, including program name and comments.
- Declarations that identify the inputs and outputs of the logic functions to be performed.
- Statements that specify the logic functions to be performed.
- Usually, a declaration of the type of PLD or other targeted device in which the specified logic functions are to be performed.
- Usually, "test vectors" that specify the logic functions' expected outputs for certain inputs.

ABEL language processor

ABEL compiler

ABEL is supported by an *ABEL language processor*, which we'll simply call an *ABEL compiler*. The compiler's job is to translate the ABEL text file into a "fuse pattern" that can be downloaded into a physical PLD. Even though most PLDs can be physically programmed only with patterns corresponding to sum-of-products expressions, ABEL allows PLD functions to be expressed using truth tables or nested "`IF`" statements as well as by any algebraic expression format. The compiler manipulates these formats and minimizes the resulting equations to fit, if possible, into the available PLD structure.

We'll talk about PLD structures, fuse patterns, and related topics later, in \secref{PLDs} and show how to target ABEL programs to specific PLDs. In the meantime, we'll show how ABEL can be used to specify combinational logic functions without necessarily having to declare the targeted device type. Later, in \chapref{seqPLDs}, we'll do the same for sequential logic functions.

4.6.1 ABEL Program Structure

Table 4-10 shows the typical structure of an ABEL program, and Table 4-11 shows an actual program exhibiting the following language features:

identifier

`module`

- *Identifiers* must begin with a letter or underscore, may contain up to 31 letters, digits, and underscores, and are case sensitive.
- A program file begins with a *`module`* statement, which associates an identifier (`Alarm_Circuit`) with the program module. Large programs can have multiple modules, each with its own local title, declarations, and equations. Note that keywords such as "`module`" are not case sensitive.

LEGAL NOTICE ABEL (Advanced Boolean Equation Language) is a trademark of Data I/O Corporation (Redmond, WA 98073).

Table 4-10
Typical structure of an ABEL program.

```
module module name
title string
deviceID device deviceType;
pin declarations
other declarations
equations
equations
test_vectors
test vectors
end module name
```

title

- The *title* statement specifies a title string that will be inserted into the documentation files that are created by the compiler.

string

- A *string* is a series of characters enclosed by single quotes.

device

- The optional *device* declaration includes a device identifier (ALARMCKT) and a string that denotes the device type ('P16V8C' for a GAL16V8). The compiler uses the device identifier in the names of documentation files that it generates, and it uses the device type to determine whether the device can really perform the logic functions specified in the program.

comment

- *Comments* begin with a double quote and end with another double quote or the end of the line, whichever comes first.

pin declarations

- *Pin declarations* tell the compiler about symbolic names associated with the device's external pins. If the signal name is preceded with the NOT prefix (!), then the complement of the named signal will appear on the pin. Pin declarations may or may not include pin numbers; if none are given, the compiler assigns them based on the capabilities of the targeted device.

istype
com

- The *istype* keyword precedes a list of one or properties, separated by commas. This tells the compiler the type of output signal. The "com" keyword indicates a combinational output. If no istype keyword is given, the compiler generally assumes that the signal is an input unless it appears on the left-hand side of an equation, in which case it tries to figure out the output's properties from the context. For your own protection, it's best just to use the istype keyword for all outputs!

other declarations

- *Other declarations* allow the designer to define constants and expressions to improve program readability and to simplify logic design.

equations

- The *equations* statement indicates that logic equations defining output signals as functions of input signals will follow.

equations

- *Equations* are written like assignment statements in a conventional programming language. Each equation is terminated by a semicolon. ABEL uses the following symbols for logical operations:

Table 4-11 An ABEL program for the alarm circuit of Figure 4-11.

```
module Alarm_Circuit
title 'Alarm Circuit Example
J. Wakerly, Micro Systems Engineering'
ALARMCKT device 'P16V8C';

" Input pins
PANIC, ENABLEA, EXITING      pin 1, 2, 3;
WINDOW, DOOR, GARAGE         pin 4, 5, 6;
" Output pins
ALARM                        pin 11 istype 'com';

" Constant definition
X = .X.;

" Intermediate equation
SECURE = WINDOW & DOOR & GARAGE;

equations
ALARM = PANIC # ENABLEA & !EXITING & !(WINDOW & DOOR & GARAGE);

test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,    .X.,    .X.,   .X., .X.,   .X.] -> [    1];
[     0,     0,     .X.,   .X., .X.,   .X.] -> [    0];
[     0,     1,       1,   .X., .X.,   .X.] -> [    0];
[     0,     1,       0,     0, .X.,   .X.] -> [    1];
[     0,     1,       0,   .X.,   0,   .X.] -> [    1];
[     0,     1,       0,   .X., .X.,     0] -> [    1];
[     0,     1,       0,     1,   1,     1] -> [    0];

end Alarm_Circuit
```

&	AND.	& (AND)
#	OR.	# (OR)
!	NOT (used as a prefix).	! (NOT)
$	XOR.	$ (XOR)
!$	XNOR.	!$ (XNOR)

As in conventional programming languages, AND (&) has precedence over OR (#) in expressions. The *@ALTERNATE* directive can be used to make the compiler recognize an alternate set of symbols for these operations: +, *, / , :+:, and :*:, respectively. This book uses the default symbols.

@ALTERNATE

- The optional *test_vectors* statement indicates that test vectors follow.

test_vectors

- *Test vectors* associate input combinations with expected output values; they are used for simulation and testing as explained in Section 4.6.7.

test vectors

.X.

- The compiler recognizes several special constants, including *.X.*, a single bit whose value is "don't-care."

end

- The *end* statement marks the end of the module.

unclocked assignment operator, =

Equations for combinational outputs use the *unclocked assignment operator*, =. The left-hand side of an equation normally contains a signal name. The right-hand side is a logic expression, not necessarily in sum-of-products form. The signal name on the left-hand side of an equation may be optionally preceded by the NOT operator `!`; this is equivalent to complementing the right-hand side. The compiler's job is to generate a fuse pattern such that the signal named on the left-hand side realizes the logic expression on the right-hand side.

4.6.2 ABEL Compiler Operation

The program in Table 4-11 realizes the alarm function that we described on page 213. The signal named ENABLE has been coded as `ENABLEA` because `ENABLE` is a reserved word in ABEL.

Notice that not all of the equations appear under the `equations` statement. An equation for an intermediate variable, `SECURE`, appears earlier. This equation is merely a definition that associates an expression with the identifier `SECURE`. The ABEL compiler substitutes this expression for the identifier `SECURE` in every place that `SECURE` appears after its definition.

In Figure 4-19 on page 214 we realized the alarm circuit directly from the SECURE and ALARM expressions, using multiple levels of logic. The ABEL compiler doesn't use expressions to interconnect gates in this way. Rather, it "crunches" the expressions to obtain a minimal two-level sum-of-products result appropriate for realization in a PLD. Thus, when compiled, Table 4-11 should yield a result equivalent to the AND-OR circuit that we showed in Figure 4-20 on page 214, which happens to be minimal.

In fact, it does. Table 4-12 shows the synthesized equations file created by the ABEL compiler. Notice that the compiler creates equations only for the `ALARM` signal, the only output. The `SECURE` signal does not appear anywhere.

The compiler finds a minimal sum-of-products expression for both `ALARM` and its complement, `!ALARM`. As mentioned previously, many PLDs have the ability selectively to invert or not to invert their AND-OR output. The "reverse polarity equation" in Table 4-12 is a sum-of-products realization of `!ALARM`, and would be used if output inversion were selected.

In this example, the reverse-polarity equation has one less product term than the normal-polarity equation for `ALARM`, so the compiler would select this equation if the targeted device has selectable output inversion. A user can also force the compiler to use either normal or reverse polarity for a signal by including the keyword "`buffer`" or "`invert`," respectively, in the signal's `istype` property list. (With some ABEL compilers, keywords "`pos`" and "`neg`" can be used for this purpose, but see Section 4.6.6.)

Table 4-12 Synthesized equations file produced by ABEL for program in Table 4-11.

```
ABEL 6.30

Design alarmckt created Tue Nov 24 1998

Title: Alarm Circuit Example
Title: J. Wakerly, Micro Systems Engineering

 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  -----------------
   4/3       6        1     Pin   ALARM
=========
   4/3          Best P-Term Total: 3
                       Total Pins: 7
                      Total Nodes: 0
            Average P-Term/Output: 3


Equations:

ALARM = (ENABLEA & !EXITING & !DOOR
     # ENABLEA & !EXITING & !WINDOW
     # ENABLEA & !EXITING & !GARAGE
     # PANIC);


Reverse-Polarity Equations:

!ALARM = (!PANIC & WINDOW & DOOR & GARAGE
     # !PANIC & EXITING
     # !PANIC & !ENABLEA);
```

4.6.3 WHEN Statements and Equation Blocks

WHEN statement

In addition to equations, ABEL provides the *WHEN statement* as another means to specify combinational logic functions in the *equations* section of an ABEL program. Table 4-13 shows the general structure of a WHEN statement, similar to an IF statement in a conventional programming language. The ELSE clause is optional. Here *LogicExpression* is an expression which results in a value of true (1) or false (0). Either *TrueEquation* or *FalseEquation* is "executed" depending

```
WHEN LogicExpression THEN
  TrueEquation;
ELSE
  FalseEquation;
```

Table 4-13 Structure of an ABEL WHEN statement.

on the value of *LogicExpression*. But we need to be a little more precise about what we mean by “executed,” as discussed below.

In the simplest case, *TrueEquation* and the optional *FalseEquation* are assignment statements, as in the first two `WHEN` statements in Table 4-14 (for `X1` and `X2`). In this case, *LogicExpression* is logically ANDed with the right-hand side of *TrueEquation*, and the complement of *LogicExpression* is ANDed with the right-hand side of *FalseEquation*. Thus, the equations for `X1A` and `X2A` produce the same results as the corresponding `WHEN` statements but do not use `WHEN`.

Notice in the first example that `X1` appears in the *TrueEquation*, but there is no *FalseEquation*. So, what happens to `X1` when *LogicExpression* `(!A#B)` is false? You might think that `X1`’s value should be don’t-care for these input combinations, but it’s not, as explained below.

Formally, the unclocked assignment operator, `=`, specifies input combinations that should be added to the on-set for the output signal appearing on the left-hand side of the equation. An output’s on-set starts out empty, and is augmented each time that the output appears on the left-hand side of an equation. That is, the right-hand sides of all equations for the same (uncomplemented) output are ORed together. (If the output appears complemented on the left-hand side, the right-hand side is complemented before being ORed.) Thus, the value of `X1` is 1 only for the input combinations for which *LogicExpression* `(!A#B)` is true and the right-hand side of *TrueEquation* `(C&!D)` is also true.

In the second example, `X2` appears on the left-hand side of two equations, so the equivalent equation shown for `X2A` is obtained by ORing two right-hand sides after ANDing each with the appropriate condition.

The *TrueEquation* and the optional *FalseEquation* in a `WHEN` statement can be any equation. In addition, `WHEN` statements can be “nested” by using another `WHEN` statement as the *FalseEquation*. When statements are nested, all of the conditions leading to an “executed” statement are ANDed. The equation for `X3` and its `WHEN`-less counterpart for `X3A` in Table 4-14 illustrate the concept.

The *TrueEquation* can be another `WHEN` statement if it’s enclosed in braces, as shown in the `X4` example in the table. This is just one instance of the general use of braces described shortly.

Although each of our `WHEN` examples have assigned values to the same output within each part of a given `WHEN` statement, this does not have to be the case. The second-to-last `WHEN` statement in Table 4-14 is such an example.

It’s often useful to make more than one assignment in *TrueEquation* or *FalseEquation* or both. For this purpose, ABEL supports equation blocks anywhere that it supports a single equation. An *equation block* is just a sequence of statements enclosed in braces, as shown in the last `WHEN` statement in the table. The individual statements in the sequence may be simple assignment statements, or they may be `WHEN` statements or nested equation blocks. A semicolon is not used after a block’s closing brace. Just for fun, Table 4-15 shows the equations that the ABEL compiler produces for the entire example program.

equation block

Table 4-14 Examples of `WHEN` statements.

```
module WhenEx
title 'WHEN Statement Examples'

" Input pins
A, B, C, D, E, F                 pin;

" Output pins
X1, X1A, X2, X2A, X3, X3A, X4    pin istype 'com';
X5, X6, X7, X8, X9, X10          pin istype 'com';

equations

WHEN (!A # B) THEN X1 = C & !D;

X1A = (!A # B) & (C & !D);

WHEN (A & B) THEN X2 = C # D;
ELSE X2 = E # F;

X2A = (A & B) & (C # D)
    # !(A & B) & (E # F);

WHEN (A) THEN X3 = D;
ELSE WHEN (B) THEN X3 = E;
ELSE WHEN (C) THEN X3 = F;

X3A = (A) & (D)
    # !(A) & (B) & (E)
    # !(A) & !(B) & (C) & (F);

WHEN (A) THEN
  {WHEN (B) THEN X4 = D;}
ELSE X4 = E;

WHEN (A & B) THEN X5 = D;
ELSE WHEN (A # !C) THEN X6 = E;
ELSE WHEN (B # C) THEN X7 = F;

WHEN (A) THEN {
    X8 = D & E & F;
    WHEN (B) THEN X8 = 1; ELSE {X9 = D; X10 = E;}
} ELSE {
    X8 = !D # !E;
    WHEN (D) THEN X9 = 1;
    {X10 = C & D;}
}

end WhenEx
```

Table 4-15 Synthesized equations file produced by ABEL for program in Table 4-14.

```
ABEL 6.30

Design whenex created Wed Dec 2 1998

Title: WHEN Statement Examples

 P-Terms  Fan-in Fan-out Type Name
--------- ------ ------- ---- -----
   2/3       4        1   Pin  X1
   2/3       4        1   Pin  X1A
   6/3       6        1   Pin  X2
   6/3       6        1   Pin  X2A
   3/4       6        1   Pin  X3
   3/4       6        1   Pin  X3A
   2/3       4        1   Pin  X4
   1/3       3        1   Pin  X5
   2/3       4        1   Pin  X6
   1/3       3        1   Pin  X7
   4/4       5        1   Pin  X8
   2/2       3        1   Pin  X9
   2/4       5        1   Pin  X10
=========
  36/42     Best P-Term Total: 30
                   Total Pins: 19
                  Total Nodes: 0
        Average P-Term/Output: 2
```

```
Equations:

X1 = (C & !D & !A
     # C & !D & B);

X1A = (C & !D & !A
     # C & !D & B);

X2 = (D & A & B
     # C & A & B
     # !B & E
     # !A & E
     # !B & F
     # !A & F);

X2A = (D & A & B
     # C & A & B
     # !B & E
     # !A & E
     # !B & F
     # !A & F);

X3 = (C & !A & !B & F
     # !A & B & E
     # D & A);

X3A = (C & !A & !B & F
     # !A & B & E
     # D & A);

X4 = (D & A & B
     # !A & E);

X5 = (D & A & B);

X6 = (A & !B & E
     # !C & !A & E);

X7 = (C & !A & F);

X8 = (D & A & E & F
     # A & B
     # !A & !E
     # !D & !A);

X9 = (D & !A
     # D & !B);

X10 = (C & D & !A
     # A & !B & E);
```

```
Reverse-Polarity Eqns:

!X1 = (A & !B
     # D
     # !C);

!X1A = (A & !B
     # D
     # !C);

!X2 = (!C & !D & A & B
     # !B & !E & !F
     # !A & !E & !F);

!X2A = (!C & !D & A & B
     # !B & !E & !F
     # !A & !E & !F);

!X3 = (!C & !A & !B
     # !A & B & !E
     # !D & A
     # !A & !B & !F);

!X3A = (!C & !A & !B
     # !A & B & !E
     # !D & A
     # !A & !B & !F);

!X4 = (A & !B
     # !D & A
     # !A & !E);

!X5 = (!A
     # !D
     # !B);

!X6 = (A & B
     # C & !A
     # !E);

!X7 = (A
     # !C
     # !F);

!X8 = (A & !B & !F
     # D & !A & E
     # A & !B & !E
     # !D & A & !B);

!X9 = (!D
     # A & B);

!X10 = (A & B
     # !D & !A
     # !C & !A
     # A & !E);
```

```
truth_table  (input-list -> output-list)
             input-value -> output-value ;
             . . .
             input-value -> output-value ;
```

Table 4-16 Structure of an ABEL truth table.

4.6.4 Truth Tables

ABEL provides one more way to specify combinational logic functions—the *truth table*, with the general format shown in Table 4-16. The keyword *truth_table* introduces a truth table. The *input-list* and *output-list* give the names of the input signals and the outputs that they affect. Each of these lists is either a single signal name or a *set*; sets are described fully in Section 4.6.5. Following the truth-table introduction are a series of statements, each of which specifies an input value and a required output value using the "->" operator. For example, the truth table for an inverter is shown below:

truth table
truth_table
input-list
output-list
unclocked truth-table operator, ->

```
truth_table (X -> NOTX)
             0 -> 1;
             1 -> 0;
```

The list of input values does not need to be complete; only the on-set of the function needs to be specified unless don't-care processing is enabled (see Section 4.6.6). Table 4-17 shows how the prime-number detector function described on page 213 can be specified using an ABEL program. For convenience, the identifier NUM is defined as a synonym for the set of four input bits [N3,N2,N1,N0], allowing a 4-bit input value to be written as a decimal integer.

Table 4-17 An ABEL program for the prime number detector.

```
module PrimeDet
title '4-Bit Prime Number Detector'

" Input and output pins
N0, N1, N2, N3                  pin;
F                               pin istype 'com';

" Definition
NUM = [N3,N2,N1,N0];

truth_table (NUM -> F)
               1 -> 1;
               2 -> 1;
               3 -> 1;
               5 -> 1;
               7 -> 1;
              11 -> 1;
              13 -> 1;
end PrimeDet
```

Both truth tables and equations can be used within the same ABEL program. The `equations` keyword introduces a sequence of equations, while the `truth_table` keyword introduces a single truth table.

4.6.5 Ranges, Sets, and Relations

Most digital systems include buses, registers, and other circuits that handle a group of two or more signals in an identical fashion. ABEL provides several shortcuts for conveniently defining and using such signals.

range

The first shortcut is for naming similar, numbered signals. As shown in the pin definitions in Table 4-18, a *range* of signal names can be defined by stating the first and last names in the range, separated by "`..`". For example, writing "`N3..N0`" is the same as writing "`N3,N2,N1,N0`." Notice in the table that the range can be ascending or descending.

set

Next, we need a facility for writing equations more compactly when a group of signals are all handled identically, in order to reduce the chance of errors and inconsistencies. An ABEL *set* is simply a defined collection of signals that is handled as a unit. When a logical operation such as AND, OR, or assignment is applied to a set, it is applied to each element of the set.

Each set is defined at the beginning of the program by associating a set name with a bracketed list of the set elements (e.g., `N=[N3,N2,N1,N0]` in Table 4-18). The set element list may use shortcut notation (`YOUT=[Y1..Y4]`), but the element names need not be similar or have any correspondence with the set name (`COMP=[EQ,GE]`). Set elements can also be constants (`GT=[0,1]`). In any case, the number and order of elements in a set are significant, as we'll see.

Most of ABEL's operators, can be applied to sets. When an operation is applied to two or more sets, all of the sets must have the same number of elements, and the operation is applied individually to set elements in like positions, regardless of their names or numbers. Thus, the equation "`YOUT = N & M`" is equivalent to four equations:

```
Y1 = N3 & M3;
Y2 = N2 & M2;
Y3 = N1 & M1;
Y4 = N0 & M0;
```

When an operation includes both set and nonset variables, the nonset variables are combined individually with set elements in each position. Thus, the equation "`ZOUT = (SEL & N) # (!SEL & M)`" is equivalent to four equations of the form "`Zi = (SEL & Ni) # (!SEL & Mi)`" for `i` equal `0` to `3`.

relation
relational operator

Another important feature is ABEL's ability to convert "relations" into logic expressions. A *relation* is a pair of operands combined with one of the *relational operators* listed in Table 4-19. The compiler converts a relation into a logic expression that is 1 if and only if the relation is true.

The operands in a relation are treated as unsigned integers, and either operand may be an integer or a set. If the operand is a set, it is treated as an unsigned

Table 4-18 Examples of ABEL ranges, sets, and relations.

```
module SetOps
title 'Set Operation Examples'

" Input and output pins
N3..N0, M3..M0, SEL                         pin;
Y1..Y4, Z0..Z3, EQ, GE, GTR, LTH, UNLUCKY    pin istype 'com';

" Definitions
N    = [N3,N2,N1,N0];
M    = [M3,M2,M1,M0];
YOUT = [Y1..Y4];
ZOUT = [Z3..Z0];

COMP = [EQ,GE];
GT   = [ 0, 1];
LT   = [ 0, 0];

equations

YOUT = N & M;
ZOUT = (SEL & N) # (!SEL & M);
EQ = (N == M);
GE = (N >= M);
GTR = (COMP == GT);
LTH = (COMP == LT);
UNLUCKY = (N == 13) # (M == ^hD) # ((N + M) == ^b1101);

end SetOps
```

binary integer with the leftmost variable representing the most significant bit. By default, numbers in ABEL programs are assumed to be base-10. Hexadecimal and binary numbers are denoted by a prefix of "`^h`" or "`^b`," respectively, as shown in the last equation in Table 4-18.

^h hexadecimal prefix
^b binary prefix

ABEL sets and relations allow a lot of functionality to be expressed in very few lines of code. For example, the equations in Table 4-18 generate minimized equations with 69 product terms, as shown in the summary in Table 4-20.

Symbol	*Relation*
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Table 4-19 Relational operators in ABEL.

Table 4-20 Synthesized equations summary produced by ABEL for program in Table 4-18.

```
 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  -----------------
   1/2        2       1     Pin   Y1
   1/2        2       1     Pin   Y2
   1/2        2       1     Pin   Y3
   1/2        2       1     Pin   Y4
   2/2        3       1     Pin   Z0
   2/2        3       1     Pin   Z1
   2/2        3       1     Pin   Z2
   2/2        3       1     Pin   Z3
  16/8        8       1     Pin   EQ
  23/15       8       1     Pin   GE
   1/2        2       1     Pin   GTR
   1/2        2       1     Pin   LTH
  16/19       8       1     Pin   UNLUCKY
=========
  69/62          Best P-Term Total: 53
                        Total Pins: 22
                       Total Nodes: 0
             Average P-Term/Output: 4
```

*4.6.6 Don't-Care Inputs

Some versions of the ABEL compiler have a limited ability to handle don't-care inputs. As mentioned previously, ABEL equations specify input combinations that belong to the on-set of a logic function; the remaining combinations are assumed to belong to the off-set. If some input combinations can instead be assigned to the d-set, then the program may be able to use these don't-care inputs to do a better job of minimization.

The ABEL language defines two mechanisms for assigning input combinations to the d-set. In order to use either mechanism, you must include the compiler directive `@DCSET` in your program, or include "`dc`" in the `istype` property list of the outputs for which you want don't-cares to be considered.

@DCSET

dc

?= don't-care unclocked assignment operator

The first mechanism is the *don't-care unclocked assignment operator*, `?=`. This operator is used instead of = in equations to indicate that input combinations matching the right-hand side should be put into the d-set instead of the on-set. Although this operator is documented in the ABEL compiler that I use, unfortunately it is broken, so I'm not going to talk about it anymore.

The second mechanism is the truth table. When don't-care processing is enabled, any input combinations that are not explicitly listed in the truth table are put into the d-set. Thus, the prime BCD-digit detector described on page 230 can be specified in ABEL as shown in Table 4-21. A don't-care value is implied for input combinations 10–15 because these combinations do not appear in the truth table and the `@DCSET` directive is in effect.

```
module DontCare
title 'Dont Care Examples'
@DCSET

" Input and output pins
N3..N0, A, B                    pin;
F, Y                           pin istype 'com';

NUM = [N3..N0];
X = .X.;

truth_table (NUM->F)
               0->0;
               1->1;
               2->1;
               3->1;
               4->0;
               5->1;
               6->0;
               7->1;
               8->0;
               9->0;

truth_table ([A,B]->Y)
             [0,0]->0;
             [0,1]->X;
             [1,0]->X;
             [1,1]->1;

end DontCare
```

Table 4-21 ABEL program using don't-cares.

It's also possible to specify don't-care combinations explicitly, as shown in the second truth table. As introduced at the very beginning of this section, ABEL recognizes `.X.` as a special one-bit constant whose value is "don't-care." In Table 4-21, the identifier "`X`" has been equated to this constant just to make it easier to type don't-cares in the truth table. The minimized equations resulting from Table 4-21 are shown in Table 4-22. Notice that the two equations for `F` are not equal; the compiler has selected different values for the don't-cares.

```
Equations:
F = (!N2 & N1
     # !N3 & N0);
Y = (B);

Reverse-Polarity Equations:
!F = (N2 & !N0
     # N3
     # !N1 & !N0);
!Y = (!B);
```

Table 4-22 Minimized equations derived from Table 4-21.

```
test_vectors  (input-list -> output-list)
              input-value -> output-value;
              . . .
              input-value -> output-value;
```

Table 4-23 Structure of ABEL test vectors.

4.6.7 Test Vectors

ABEL programs may contain optional test vectors, as we showed in Table 4-11 on page 249. The general format of test vectors is very similar to a truth table and is shown in Table 4-23. The keyword `test_vectors` introduces a truth table. The *input-list* and *output-list* give the names of the input signals and the outputs that they affect. Each of these lists is either a single signal name or a set. Following the test-vector introduction are a series of statements, each of which specifies an input value and an expected output value using the "`->`" operator.

test_vectors

input-list
output-list

ABEL test vectors have two main uses and purposes:

1. After the ABEL compiler translates the program into "fuse pattern" for a particular device, it simulates the operation of the final programmed device by applying the test-vector inputs to a software model of the device and comparing its outputs with the corresponding test-vector outputs. The designer may specify a series of test vectors in order to double-check that device will behave as expected for some or all input combinations.
2. After a PLD is physically programmed, the programming unit applies the test-vector inputs to the physical device and compares the device outputs with the corresponding test-vector outputs. This is done to check for correct device programming and operation.

Unfortunately, ABEL test vectors seldom do a very good job at either one of these tasks, as we'll explain.

The test vectors from Table 4-11 are repeated in Table 4-24, except that for readability we've assumed that the identifier `X` has been equated to the don't-care constant `.X.`, and we've added comments to number the test vectors.

Table 4-24 actually appears to be a pretty good set of test vectors. From the designer's point of view, these vectors fully cover the expected operation of the alarm circuit, as itemized vector-by-vector below:

1. If `PANIC` is 1, then the alarm output (`F`) should be on regardless of the other input values. All of the remaining vectors cover cases where `PANIC` is 0.
2. If the alarm is not enabled, then the output should be off.
3. If the alarm is enabled but we're exiting, then the output should be off.

4-6. If the alarm is enabled and we're not exiting, then the output should be on if any of the sensor signals `WINDOW`, `DOOR`, or `GARAGE` is 0.

7. If the alarm is enabled, we're not exiting, and all of the sensor signals are 1, then the output should be off.

```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,       X,       X,      X,    X,      X] -> [    1];   1
[     0,       0,       X,      X,    X,      X] -> [    0];   2
[     0,       1,       1,      X,    X,      X] -> [    0];   3
[     0,       1,       0,      0,    X,      X] -> [    1];   4
[     0,       1,       0,      X,    0,      X] -> [    1];   5
[     0,       1,       0,      X,    X,      0] -> [    1];   6
[     0,       1,       0,      1,    1,      1] -> [    0];   7
```

Table 4-24 Test vectors for the alarm circuit program in Table 4-11.

The problem is that ABEL doesn't handle don't-cares in test-vector inputs the way that it should. For example, by all rights, test vector 1 should test 32 distinct input combinations corresponding to all 32 possible combinations of don't-care inputs ENABLEA, EXITING, WINDOW, DOOR, and GARAGE. But it doesn't. In this situation, the ABEL compiler interprets "don't care" as "the user doesn't care what input value I use," and it just assigns 0 to all don't-care inputs in a test vector. In this example, you could have erroneously written the output equation as "F = PANIC & !ENABLEA # ENABLEA & ..."; the test vectors would still pass even though the panic button would work only when the system is disabled.

The second use of test vectors is in physical device testing. Most physical defects in logic devices can be detected using the *single stuck-at fault model*, which assumes that any physical defect is equivalent to having a single gate input or output stuck at a logic 0 or 1 value. Just putting together a set of test vectors that seems to exercise a circuit's functional specifications, as we did in Table 4-24, doesn't guarantee that all single stuck-at faults can be detected. The test vectors have to be chosen so that every possible stuck-at fault causes an incorrect value at the circuit output for some test-vector input combination.

single stuck-at fault model

Table 4-25 shows a complete set of test vectors for the alarm circuit when it is realized as a two-level sum-of-products circuit. The first four vectors check for stuck-at-1 faults on the OR gate, and the last three check for stuck-at-0 faults on the AND gates; it turns out that this is sufficient to detect all single stuck-at faults. If you know something about fault testing you can generate test vectors for small circuits by hand (as I did in this example), but most designers use automated third-party tools to create high-quality test vectors for their PLD designs.

```
test_vectors
([PANIC,ENABLEA,EXITING,WINDOW,DOOR,GARAGE] -> [ALARM])
[     1,       0,       1,      1,    1,      1] -> [    1];   1
[     0,       1,       0,      0,    1,      1] -> [    1];   2
[     0,       1,       0,      1,    0,      1] -> [    1];   3
[     0,       1,       0,      1,    1,      0] -> [    1];   4
[     0,       0,       0,      0,    0,      0] -> [    0];   5
[     0,       1,       1,      0,    0,      0] -> [    0];   6
[     0,       1,       0,      1,    1,      1] -> [    0];   7
```

Table 4-25 Single-stuck-at-fault test vectors for the minimal sum-of-products realization of the alarm circuit.

4.7 VHDL

References

A historical description of Boole's development of "the science of Logic" appears in *The Computer from Pascal to von Neumann* by Herman H. Goldstine (Princeton University Press, 1972). Claude E. Shannon showed how Boole's work could be applied to logic circuits in "A Symbolic Analysis of Relay and Switching Circuits" (*Trans. AIEE*, Vol. 57, 1938, pp. 713–723).

Although the two-valued Boolean algebra is the basis for switching algebra, a Boolean algebra need not have only two values. Boolean algebras with 2^n values exist for every integer n; for example, see *Discrete Mathematical Structures and Their Applications* by Harold S. Stone (SRA, 1973). Such algebras may be formally defined using the so-called *Huntington postulates* devised by E. V. Huntington in 1907; for example, see *Digital Design* by M. Morris Mano (Prentice Hall, 1984). A mathematician's development of Boolean algebra based on a more modern set of postulates appears in *Modern Applied Algebra* by G. Birkhoff and T. C. Bartee (McGraw-Hill, 1970). Our engineering-style, "direct" development of switching algebra follows that of Edward J. McCluskey in his *Introduction to the Theory of Switching Circuits* (McGraw-Hill, 1965) and *Logic Design Principles* (Prentice Hall, 1986).

Huntington postulates

The prime implicant theorem was proved by W. V. Quine in "The Problem of Simplifying Truth Functions" (*Am. Math. Monthly*, Vol. 59, No. 8, 1952, pp. 521–531). In fact it is possible to prove a more general prime implicant theorem showing that there exists at least one minimal sum that is a sum of prime implicants even if we remove the constraint on the number of literals in the definition of "minimal."

A graphical method for simplifying Boolean functions was proposed by E. W. Veitch in "A Chart Method for Simplifying Boolean Functions" (*Proc. ACM*, May 1952, pp. 127–133). His *Veitch diagram*, shown in Figure 4-50, actually reinvented a chart proposed by an English archaeologist, A. Marquand ("On Logical Diagrams for *n* Terms," *Philosophical Magazine* XII, 1881, pp. 266–270). The Veitch diagram or Marquand chart uses "natural" binary count-

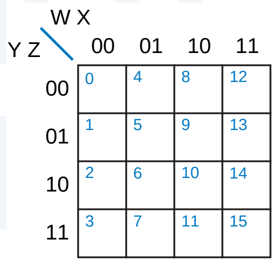


Figure 4-50
A 4-variable Veitch diagram or Marquand chart.

ing order for its rows and columns, with the result that some adjacent rows and columns differ in more than one value, and product terms do not always cover adjacent cells. M. Karnaugh showed how to fix the problem in “A Map Method for Synthesis of Combinational Logic Circuits” (*Trans. AIEE, Comm. and Electron.*, Vol. 72, Part I, November 1953, pp. 593–599). On the other hand, George J. Klir, in his book *Introduction to the Methodology of Switching Circuits*, claims that binary counting order is just as good as, perhaps better than Karnaugh-map order for minimizing logic functions.

At this point, the Karnaugh vs. Veitch argument is of course irrelevant, because no one draws charts any more to minimize logic circuits. Instead, we use computer programs running logic minimization algorithms. The first of such algorithms was described by W. V. Quine in “A Way to Simplify Truth Functions” (*Am. Math. Monthly*, Vol. 62, No. 9, 1955, pp. 627–631) and modified by E. J. McCluskey in “Minimization of Boolean Functions” (*Bell Sys. Tech. J.*, Vol. 35, No. 5, November 1956, pp. 1417–1444). The Quine-McCluskey algorithm is fully described in McCluskey’s books cited earlier.

McCluskey’s 1965 book also covers the iterative consensus algorithm for finding prime implicants, and proves that it works. The starting point for this algorithm is a sum-of-products expression, or equivalently, a list of cubes. The product terms *need not* be minterms or prime implicants, but *may* be either or anything in between. In other words, the cubes in the list may have any and all dimensions, from 0 to *n* in an *n*-variable function. Starting with the list of cubes, the algorithm generates a list of all the prime-implicant cubes of the function, without ever having to generate a full minterm list.

The iterative consensus algorithm was first published by T. H. Mott, Jr., in “Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants” (*IRE Trans. Electron. Computers*, Vol. EC-9, No. 2, 1960, pp. 245–252). A generalized consensus algorithm was published by Pierre Tison in “Generalization of Consensus Theory and Application to the Minimization of Boolean Functions” (*IEEE Trans. Electron. Computers*, Vol. EC-16, No. 4, 1967, pp. 446–456). All of these algorithms are described by Thomas Downs in *Logic Design with Pascal* (Van Nostrand Reinhold, 1988).

As we explained in Section 4.4.4, the huge number of prime implicants in some logic functions makes it impractical or impossible deterministically to find them all or select a minimal cover. However, efficient heuristic methods can find solutions that are close to minimal. The Espresso-II method is described in *Logic Minimization Algorithms for VLSI Synthesis* by R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli (Kluwer Academic Publishers, 1984). The more recent Espresso-MV and Espresso-EXACT algorithms are described in “Multiple-Valued Minimization for PLA Optimization” by R. L. Rudell and A. Sangiovanni-Vincentelli (*IEEE Trans. CAD*, Vol. CAD-6, No. 5, 1987, pp. 727–750).

In this chapter we described a map method for finding static hazards in two-level AND-OR and OR-AND circuits, but any combinational circuit can be analyzed for hazards. In both his 1965 and 1986 books, McCluskey defines the *0-set* and *1-sets* of a circuit and shows how they can be used to find static hazards. He also defines *P-sets* and *S-sets* and shows how they can be used to find dynamic hazards.

0-set
1-set
P-set
S-set

Many deeper and varied aspects of switching theory have been omitted from this book, but have been beaten to death in other books and literature. A good starting point for an academic study of classical switching theory is Zvi Kohavi's book, *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill, 1978), which includes material on set theory, symmetric networks, functional decomposition, threshold logic, fault detection, and path sensitization. Another area of great academic interest (but little commercial activity) is nonbinary *multiple-valued logic*, in which each signal line can take on more than two values. In his 1986 book, McCluskey gives a good introduction to multiple-valued logic, explaining its pros and cons and why it has seen little commercial development.

multiple-valued logic

Over the years, I've struggled to find a readily accessible and definitive reference on the ABEL language, and I've finally found it—Appendix A of *Digital Design Using ABEL*, by David Pellerin and Michael Holley (Prentice Hall, 1994). It makes sense that this would be the definitive work—Pellerin and Holley invented the language and wrote the original compiler code!

All of the ABEL and VHDL examples in this chapter and throughout the text were compiled and in most cases simulated using Foundation 1.5 Student Edition software from Xilinx, Inc. (San Jose, CA 95124, `www.xilinx.com`). This package integrates a schematic editor, HDL editor, compilers for ABEL, VHDL and Verilog, and a simulator from Aldec, Inc. (Henderson, NV 89014, `www.aldec.com`) along with Xilinx' own specialized tools for CPLD and FPGA design and programming. This software package includes an excellent on-line help system, including reference manuals for both ABEL and VHDL.

We briefly discussed device testing in the context of ABEL test vectors. There is a large, well-established body of literature on digital device testing, and a good starting point for study is McCluskey's 1986 book. Generating a set of test vectors that completely tests a large circuit such as a PLD is a task best left to a program. At least one company's entire business is focused on programs that automatically create test vectors for PLD testing (ACUGEN Software, Inc., Nashua, NH 03063, `www.acugen.com`).

Drill Problems

4.1 Using variables NERD, DESIGNER, FAILURE, and STUDIED, write a boolean expression that is 1 for successful designers who never studied and for nerds who studied all the time.

4.2 Prove theorems T2–T5 using perfect induction.

4.3 Prove theorems T1′–T3′ and T5′ using perfect induction.

4.4 Prove theorems T6–T9 using perfect induction.

4.5 According to DeMorgan's theorem, the complement of $X + Y \cdot Z$ is $X' \cdot Y' + Z'$. Yet both functions are 1 for $XYZ = 110$. How can both a function and its complement be 1 for the same input combination? What's wrong here?

4.6 Use the theorems of switching algebra to simplify each of the following logic functions:

(a) $F = W \cdot X \cdot Y \cdot Z \cdot (W \cdot X \cdot Y \cdot Z' + W \cdot X' \cdot Y \cdot Z + W' \cdot X \cdot Y \cdot Z + W \cdot X \cdot Y' \cdot Z)$

(b) $F = A \cdot B + A \cdot B \cdot C' \cdot D + A \cdot B \cdot D \cdot E' + A \cdot B \cdot C' \cdot E + C' \cdot D \cdot E$

(c) $F = M \cdot N \cdot O + Q' \cdot P' \cdot N' + P \cdot R \cdot M + Q' \cdot O \cdot M \cdot P' + M \cdot R$

4.7 Write the truth table for each of the following logic functions:

(a) $F = X' \cdot Y + X' \cdot Y' \cdot Z$ (b) $F = W' \cdot X + Y' \cdot Z' + X' \cdot Z$

(c) $F = W + X' \cdot (Y' + Z)$ (d) $F = A \cdot B + B' \cdot C + C' \cdot D + D' \cdot A$

(e) $F = V \cdot W + X' \cdot Y' \cdot Z$ (f) $F = (A' + B' + C \cdot D) \cdot (B + C' + D' \cdot E')$

(g) $F = (W \cdot X)' \cdot (Y' + Z')'$ (h) $F = (((A + B)' + C')' + D)'$

(i) $F = (A' + B + C) \cdot (A + B' + D') \cdot (B + C' + D') \cdot (A + B + C + D)$

4.8 Write the truth table for each of the following logic functions:

(a) $F = X' \cdot Y' \cdot Z' + X \cdot Y \cdot Z + X \cdot Y' \cdot Z$ (b) $F = M' \cdot N' + M \cdot P + N' \cdot P$

(c) $F = A \cdot B + A \cdot B' \cdot C' + A' \cdot B \cdot C$ (d) $F = A' \cdot B \cdot (C \cdot B \cdot A' + B \cdot C')$

(e) $F = X \cdot Y \cdot (X' \cdot Y \cdot Z + X \cdot Y' \cdot Z + X \cdot Y \cdot Z' + X' \cdot Y' \cdot Z)$ (f) $F = M \cdot N + M' \cdot N' \cdot P'$

(g) $F = (A + A') \cdot B + B \cdot A \cdot C' + C \cdot (A + B') \cdot (A' + B)$ (h) $F = X \cdot Y' + Y \cdot Z + Z' \cdot X$

4.9 Write the canonical sum and product for each of the following logic functions:

(a) $F = \Sigma_{X,Y}(1,2)$ (b) $F = \prod_{A,B}(0,1,2)$

(c) $F = \Sigma_{A,B,C}(2,4,6,7)$ (d) $F = \prod_{W,X,Y}(0,1,3,4,5)$

(e) $F = X + Y' \cdot Z'$ (f) $F = V' + (W' \cdot X)'$

4.10 Write the canonical sum and product for each of the following logic functions:

(a) $F = \Sigma_{X,Y,Z}(0,3)$ (b) $F = \prod_{A,B,C}(1,2,4)$

(c) $F = \Sigma_{A,B,C,D}(1,2,5,6)$ (d) $F = \prod_{M,N,P}(0,1,3,6,7)$

(e) $F = X' + Y \cdot Z' + Y \cdot Z'$ (f) $F = A'B + B'C + A$

4.11 If the canonical sum for an *n*-input logic function is also a minimal sum, how many literals are in each product term of the sum? Might there be any other minimal sums in this case?

4.12 Give two reasons why the cost of inverters is not included in the definition of "minimal" for logic minimization.

4.13 Using Karnaugh maps, find a minimal sum-of-products expression for each of the following logic functions. Indicate the distinguished 1-cells in each map.

(a) $F = \Sigma_{X,Y,Z}(1,3,5,6,7)$ (b) $F = \Sigma_{W,X,Y,Z}(1,4,5,6,7,9,14,15)$

(c) $F = \prod_{W,X,Y}(0,1,3,4,5)$ (d) $F = \Sigma_{W,X,Y,Z}(0,2,5,7,8,10,13,15)$

(e) $F = \prod_{A,B,C,D}(1,7,9,13,15)$ (f) $F = \Sigma_{A,B,C,D}(1,4,5,7,12,14,15)$

4.14 Find a minimal product-of-sums expression for each function in Drill 4.13 using the method of Section 4.3.6.

4.15 Find a minimal product-of-sums expression for the function in each of the following figures and compare its cost with the previously found minimal sum-of-products expression: (a) Figure 4-27; (b) Figure 4-29; (c) Figure 4-33.

4.16 Using Karnaugh maps, find a minimal sum-of-products expression for each of the following logic functions. Indicate the distinguished 1-cells in each map.

(a) $F = \Sigma_{A,B,C}(0,1,2,4)$ (b) $F = \Sigma_{W,X,Y,Z}(1,4,5,6,11,12,13,14)$

(c) $F = \prod_{A,B,C}(1,2,6,7)$ (d) $F = \Sigma_{W,X,Y,Z}(0,1,2,3,7,8,10,11,15)$

(e) $F = \Sigma_{W,X,Y,X}(1,2,4,7,8,11,13,14)$ (f) $F = \prod_{A,B,C,D}(1,3,4,5,6,7,9,12,13,14)$

4.17 Find a minimal product-of-sums expression for each function in Drill 4.16 using the method of Section 4.3.6.

4.18 Find the complete sum for the logic functions in Drill 4.16(d) and (e).

4.19 Using Karnaugh maps, find a minimal sum-of-products expression for each of the following logic functions. Indicate the distinguished 1-cells in each map.

(a) $F = \Sigma_{W,X,Y,Z}(0,1,3,5,14) + d(8,15)$ (b) $F = \Sigma_{W,X,Y,Z}(0,1,2,8,11) + d(3,9,15)$

(c) $F = \Sigma_{A,B,C,D}(1,5,9,14,15) + d(11)$ (d) $F = \Sigma_{A,B,C,D}(1,5,6,7,9,13) + d(4,15)$

(e) $F = \Sigma_{W,X,Y,Z}(3,5,6,7,13) + d(1,2,4,12,15)$

4.20 Repeat Drill 4.19, finding a minimal product-of-sums expression for each logic function.

4.21 For each logic function in the two preceding exercises, determine whether the minimal sum-of-products expression equals the minimal product-of-sums expression. Also compare the circuit cost for realizing each of the two expressions.

4.22 For each of the following logic expressions, find all of the static hazards in the corresponding two-level AND-OR or OR-AND circuit, and design a hazard-free circuit that realizes the same logic function.

(a) $F = W \cdot X + W'Y'$ (b) $F = W \cdot X' \cdot Y' + X \cdot Y' \cdot Z + X \cdot Y$

(c) $F = W' \cdot Y + X' \cdot Y' + W \cdot X \cdot Z$ (d) $F = W' \cdot X + Y' \cdot Z + W \cdot X \cdot Y \cdot Z + W \cdot X' \cdot Y \cdot$

(e) $F = (W + X + Y) \cdot (X' + Z')$ (f) $F = (W + Y'+Z') \cdot (W' + X' + Z') \cdot (X'+Y+Z)$

(g) $F = (W + Y + Z') \cdot (W + X' + Y + Z) \cdot (X' + Y') \cdot (X + Z)$

Exercises

4.23 Design a non-trivial-looking logic circuit that contains a feedback loop but whose output depends only on its current input.

4.24 Prove the combining theorem T10 without using perfect induction, but assuming that theorems T1–T9 and T1′–T9′ are true.

4.25 Show that the combining theorem, T10, is just a special case of consensus (T11) used with covering (T9).

4.26 Prove that (X + Y′) · Y = X · Y *without* using perfect induction. You may assume that theorems T1–T11 and T1′–T11′ are true.

4.27 Prove that (X+Y) · (X′+ Z) = X · Z + X′ · Y *without* using perfect induction. You may assume that theorems T1–T11 and T1′–T11′ are true.

4.28 Show that an *n*-input AND gate can be replaced by *n*–1 2-input AND gates. Can the same statement be made for NAND gates? Justify your answer.

4.29 How many physically different ways are there to realize V · W · X · Y · Z using four 2-input AND gates (4/4 of a 74LS08)? Justify your answer.

4.30 Use switching algebra to prove that tying together two inputs of an *n* + 1-input AND or OR gate gives it the functionality of an *n*-input gate.

4.31 Prove DeMorgan's theorems (T13 and T13′) using finite induction.

4.32 Which logic symbol more closely approximates the internal realization of a TTL NOR gate, Figure 4-4(c) or (d)? Why?

4.33 Use the theorems of switching algebra to rewrite the following expression using as few inversions as possible (complemented parentheses are allowed):

$$B' \cdot C + A \cdot C \cdot D' + A' \cdot C + E \cdot B' + E \cdot (A + C) \cdot (A' + D')$$

4.34 Prove or disprove the following propositions:

(a) Let A and B be switching-algebra *variables*. Then A · B = 0 and A + B = 1 implies that A = B′.

(b) Let X and Y be switching-algebra *expressions*. Then X · Y = 0 and X + Y = 1 implies that X = Y′.

4.35 Prove Shannon's expansion theorems. (*Hint:* Don't get carried away; it's easy.)

4.36 Shannon's expansion theorems can be generalized to "pull out" not just one but *i* variables so that a logic function can be expressed as a sum or product of 2^i terms. State the generalized Shannon expansion theorems.

generalized Shannon expansion theorems

4.37 Show how the generalized Shannon expansion theorems lead to the canonical sum and canonical product representations of logic functions.

4.38 An *Exclusive OR (XOR) gate* is a 2-input gate whose output is 1 if and only if exactly one of its inputs is 1. Write a truth table, sum-of-products expression, and corresponding AND-OR circuit for the Exclusive OR function.

Exclusive OR (XOR) gate

4.39 From the point of view of switching algebra, what is the function of a 2-input XOR gate whose inputs are tied together? How might the output behavior of a real XOR gate differ?

4.40 After completing the design and fabrication of a digital system, a designer finds that one more inverter is required. However, the only spare gates in the system are a 3-input OR, a 2-input AND, and a 2-input XOR. How should the designer realize the inverter function without adding another IC?

complete set

4.41 Any set of logic-gate types that can realize any logic function is called a *complete set* of logic gates. For example, 2-input AND gates, 2-input OR gates, and inverters are a complete set, because any logic function can be expressed as a sum of products of variables and their complements, and AND and OR gates with any number of inputs can be made from 2-input gates. Do 2-input NAND gates form a complete set of logic gates? Prove your answer.

4.42 Do 2-input NOR gates form a complete set of logic gates? Prove your answer.

4.43 Do 2-input XOR gates form a complete set of logic gates? Prove your answer.

4.44 Define a two-input gate, other than NAND, NOR, or XOR, that forms a complete set of logic gates if the constant inputs 0 and 1 are allowed. Prove your answer.

BUT
BUT gate

4.45 Some people think that there are *four* basic logic functions, AND, OR, NOT, and *BUT*. Figure X4.45 is a possible symbol for a 4-input, 2-output *BUT gate*. Invent a useful, nontrivial function for the BUT gate to perform. The function should have something to do with the name (BUT). Keep in mind that, due to the symmetry of the symbol, the function should be symmetric with respect to the A and B inputs of each section and with respect to sections 1 and 2. Describe your BUT's function and write its truth table.

4.46 Write logic expressions for the Z1 and Z2 outputs of the BUT gate you designed in the preceding exercise, and draw a corresponding logic diagram using AND gates, OR gates, and inverters.

4.47 Most students have no problem using theorem T8 to "multiply out" logic expressions, but many develop a mental block if they try to use theorem T8′ to "add out" a logic expression. How can duality be used to overcome this problem?

4.48 How many different logic functions are there of *n* variables?

4.49 How many different 2-variable logic functions F(X,Y) are there? Write a simplified algebraic expression for each of them.

self-dual logic function
$\oplus$

4.50 A *self-dual logic function* is a function F such that $F = F^D$. Which of the following functions are self-dual? (The symbol $\oplus$ denotes the Exclusive OR (XOR) operation.)

(a) $F = X$

(b) $F = \Sigma_{X,Y,Z}(0,3,5,6)$

(c) $F = X \cdot Y' + X' \cdot Y$

(d) $F = W \cdot (X \oplus Y \oplus Z) + W' \cdot (X \oplus Y \oplus Z)'$

(e) A function F of 7 variables such that F = 1 if and only if 4 or more of the variables are 1

(f) A function F of 10 variables such that F = 1 if and only if 5 or more of the variables are 1

4.51 How many self-dual logic functions of *n* input variables are there? (*Hint:* Consider the structure of the truth table of a self-dual function.)

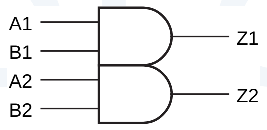


Figure X4.45

4.52 Prove that any n-input logic function $F(X_1,\ldots,X_n)$ that can be written in the form $F = X_1 \cdot G(X_2,\ldots,X_n) + X_1' \cdot G^D(X_2,\ldots,X_n)$ is self-dual.

4.53 Assuming that an inverting gate has a propagation delay of 5 ns, and a noninverting gate has a propagation delay of 8 ns, compare the speeds of the circuits in Figure 4-24(a), (c), and (d).

4.54 Find the minimal product-of-sums expressions for the logic functions in Figures 4-27 and 4-29.

4.55 Use switching algebra to show that the logic functions obtained in Exercise 4.54 equal the AND-OR functions obtained in Figures 4-27 and 4-29.

4.56 Determine whether the product-of-sums expressions obtained by “adding out” the minimal sums in Figure 4-27 and 4-29 are minimal.

4.57 Prove that the rule for combining 2^i 1-cells in a Karnaugh map is true, using the axioms and theorems of switching algebra.

4.58 An *irredundant sum* for a logic function F is a sum of prime implicants for F such that if any prime implicant is deleted, the sum no longer equals F. This sounds a lot like a minimal sum, but an irredundant sum is not necessarily minimal. For example, the minimal sum of the function in Figure 4-35 has only three product terms, but there is an irredundant sum with four product terms. Find the irredundant sum and draw a map of the function, circling only the prime implicants in the irredundant sum.

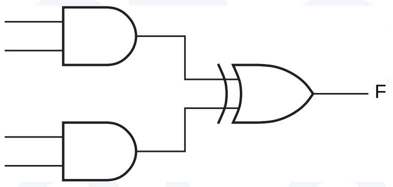
irredundant sum

4.59 Find another logic function in Section 4.3 that has one or more nonminimal irredundant sums, and draw its map, circling only the prime implicants in the irredundant sum.

4.60 Derive the minimal product-of-sums expression for the prime BCD-digit detector function of Figure 4-37. Determine whether or not the expression algebraically equals the minimal sum-of-products expression and explain your result.

4.61 Draw a Karnaugh map and assign variables to the inputs of the AND-XOR circuit in Figure X4.61 so that its output is $F = \Sigma_{W,X,Y,Z}(6,7,12,13)$. Note that the output gate is a 2-input XOR rather than an OR.

Figure X4.61



4.62 The text indicates that a truth table or equivalent is the starting point for traditional combinational minimization methods. A Karnaugh map itself contains the same information as a truth table. Given a sum-of-products expression, it is possible to write the 1s corresponding to each product term directly on the map without developing an explicit truth table or minterm list, and then proceed with

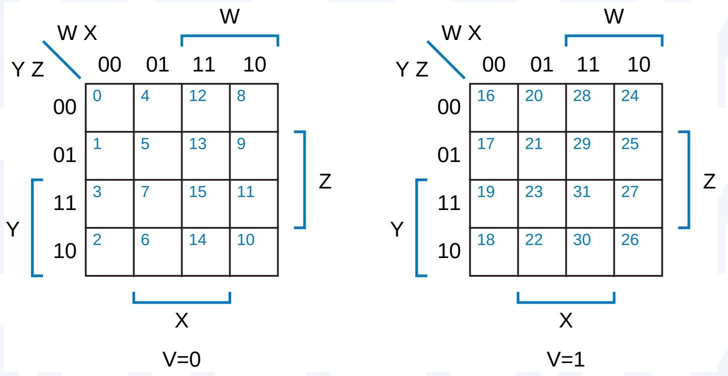


Figure X

the map minimization procedure. Find a minimal sum-of-products expression for each of the following logic functions in this way:

(a) $F = X' \cdot Z + X \cdot Y + X \cdot Y' \cdot Z$

(b) $F = A' \cdot C' \cdot D + B' \cdot C \cdot D + A \cdot C' \cdot D + B \cdot C \cdot D$

(c) $F = W \cdot X \cdot Z' + W \cdot X' \cdot Y \cdot Z + X \cdot Z$

(d) $F = (X' + Y') \cdot (W' + X' + Y) \cdot (W' + X + Z)$

(e) $F = A \cdot B \cdot C' \cdot D' + A' \cdot B \cdot C' + A \cdot B \cdot D + A' \cdot C \cdot D + B \cdot C \cdot D'$

4.63 Repeat Exercise 4-60, finding a minimal product-of-sums expression for each logic function.

5-variable Karnaugh map

4.64 A Karnaugh map for a 5-variable function can be drawn as shown in Figure X4.64. In such a map, cells that occupy the same relative position in the V = 0 and V = 1 submaps are considered to be adjacent. (Many worked examples of 5-variable Karnaugh maps appear in Sections \ref{synD} and~\ref{synJK}.) Find a minimal sum-of-products expression for each of the following functions using a 5-variable map:

(a) $F = \Sigma_{V,W,X,Y,Z}(5,7,13,15,16,20,25,27,29,31)$

(b) $F = \Sigma_{V,W,X,Y,Z}(0,7,8,9,12,13,15,16,22,23,30,31)$

(c) $F = \Sigma_{V,W,X,Y,Z}(0,1,2,3,4,5,10,11,14,20,21,24,25,26,27,28,29,30)$

(d) $F = \Sigma_{V,W,X,Y,Z}(0,2,4,6,7,8,10,11,12,13,14,16,18,19,29,30)$

(e) $F = \prod_{V,W,X,Y,Z}(4,5,10,12,13,16,17,21,25,26,27,29)$

(f) $F = \Sigma_{V,W,X,Y,Z}(4,6,7,9,11,12,13,14,15,20,22,25,27,28,30)+d(1,5,29,31)$

4.65 Repeat Exercise 4.64, finding a minimal product-of-sums expression for each logic function.

6-variable Karnaugh map

4.66 A Karnaugh map for a 6-variable function can be drawn as shown in Figure X4.66. In such a map, cells that occupy the same relative position in adjacent submaps are considered to be adjacent. Minimize the following functions using 6-variable maps:

(a) $F = \Sigma_{U,V,W,X,Y,Z}(1,5,9,13,21,23,29,31,37,45,53,61)$

(b) $F = \Sigma_{U,V,W,X,Y,Z}(0,4,8,16,24,32,34,36,37,39,40,48,50,56)$

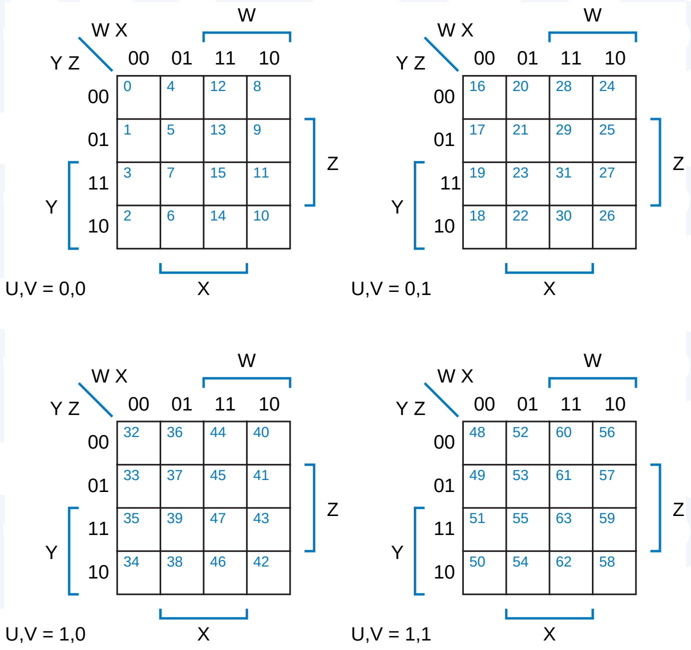


Figure X4.66

(c) $F = \Sigma_{U,V,W,X,Y,Z}(2,4,5,6,12\text{–}21,28\text{–}31,34,38,50,51,60\text{–}63)$

4.67 A 3-bit "comparator" circuit receives two 3-bit numbers, $P = P_2P_1P_0$ and $Q = Q_2Q_1Q_0$. Design a minimal sum-of-products circuit that produces a 1 output if and only if $P > Q$.

4.68 Find minimal multiple-output sum-of-products expressions for $F = \Sigma_{X,Y,Z}(0,1,2)$, $G = \Sigma_{X,Y,Z}(1,4,6)$, and $H = \Sigma_{X,Y,Z}(0,1,2,4,6)$.

4.69 Prove whether or not the following expression is a minimal sum. Do it the easiest way possible (algebraically, not using maps).

$$F = T' \cdot U \cdot V \cdot W \cdot X + T' \cdot U \cdot V' \cdot X \cdot Z + T' \cdot U \cdot W \cdot X \cdot Y' \cdot Z$$

4.70 There are $2n$ m-subcubes of an n-cube for the value $m = n - 1$. Show their text representations and the corresponding product terms. (You may use ellipses as required, e.g., 1, 2, …, n.)

4.71 There is just one m-subcube of an n-cube for the value $m = n$; its text representation is xx…xx. Write the product term corresponding to this cube.

4.72 The C program in Table 4-9 uses memory inefficiently because it allocates memory for a maximum number of cubes at each level, even if this maximum is never used. Redesign the program so that the `cubes` and `used` arrays are one-dimensional arrays, and each level uses only as many array entries as needed. (*Hint:* You can still allocate cubes sequentially, but keep track of the starting point in the array for each level.)

4.73 As a function of m, how many times is each distinct m-cube rediscovered in Table 4-9, only to be found in the inner loop and thrown away? Suggest some ways to eliminate this inefficiency.

4.74 The third `for`-loop in Table 4-9 tries to combine all *m*-cubes at a given level with all other *m*-cubes at that level. In fact, only *m*-cubes with x's in the same positions can be combined, so it is possible to reduce the number of loop iterations by using a more sophisticated data structure. Design a data structure that segregates the cubes at a given level according to the position of their x's, and determine the maximum size required for various elements of the data structure. Rewrite Table 4-9 accordingly.

4.75 Estimate whether the savings in inner-loop iterations achieved in Exercise 4.75 outweighs the overhead of maintaining a more complex data structure. Try to make reasonable assumptions about how cubes are distributed at each level, and indicate how your results are affected by these assumptions.

4.76 Optimize the `Oneones` function in Table 4-8. An obvious optimization is to drop out of the loop early, but other optimizations exist that eliminate the `for` loop entirely. One is based on table look-up and another uses a tricky computation involving complementing, Exclusive ORing, and addition.

4.77 Extend the C program in Table 4-9 to handle don't-care conditions. Provide another data structure, `dc[MAX_VARS+1][MAX_CUBES]`, that indicates whether a given cube contains only don't-cares, and update it as cubes are read and generated.

4.78 (*Hamlet circuit*.) Complete the timing diagram and explain the function of the circuit in Figure X4.78. Where does the circuit get its name?

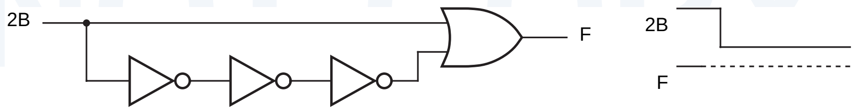


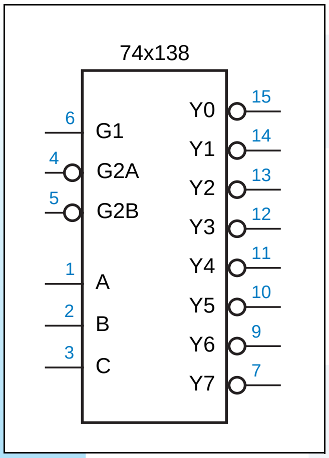
Figure X

4.79 Prove that a two-level AND-OR circuit corresponding to the complete sum of a logic function is always hazard free.

4.80 Find a four-variable logic function whose minimal sum-of-products realization is not hazard free, but where there exists a hazard-free sum-of-products realization with fewer product terms than the complete sum.

4.81 Starting with the `WHEN` statements in the ABEL program in Table 4-14, work out the logic equations for variables `X4` through `X10` in the program. Explain any discrepancies between your results and the equations in Table 4-15.

4.82 Draw a circuit diagram corresponding to the minimal two-level sum-of-products equations for the alarm circuit, as given in Table 4-12. On each inverter, AND gate, and OR gate input and output, write a pair of numbers (*t0*,*t1*), where *t0* is the test number from Table 4-25 that detects a stuck-at-0 fault on that line, and *t1* is the test number that detects a stuck-at-1 fault.



chapter 5

Combinational Logic Design Practices

The preceding chapter described the theoretical principles used in combinational logic design. In this chapter, we'll build on that foundation and describe many of the devices, structures, and methods used by engineers to solve practical digital design problems.

A practical combinational circuit may have dozens of inputs and outputs and could require hundreds, thousands, even millions of terms to describe as a sum of products, and *billions and billions* of rows to describe in a truth table. Thus, most real combinational logic design problems are too large to solve by "brute-force" application of theoretical techniques.

But wait, you say, how could any human being conceive of such a complex logic circuit in the first place? The key is structured thinking. A complex circuit or system is conceived as a collection of smaller subsystems, each of which has a much simpler description.

In combinational logic design, there are several straightforward structures—decoders, multiplexers, comparators, and the like—that turn up quite regularly as building blocks in larger systems. The most important of these structures are described in this chapter. We describe each structure generally and then give examples and applications using 74-series components, ABEL, and VHDL.

Before launching into these combinational building blocks, we need to discuss several important topics. The first topic is documentation standards

THE IMPORTANCE OF 74-SERIES LOGIC

Later in this chapter, we'll look at commonly used 74-series ICs that perform well-structured logic functions. These parts are important building blocks in a digital designer's toolbox because their level of functionality often matches a designer's level of thinking when partitioning a large problem into smaller chunks.

Even when you design for PLDs, FPGAs, or ASICs, understanding 74-series MSI functions is important. In PLD-based design, standard MSI functions can be used as a starting point for developing logic equations for more specialized functions. And in FPGA and ASIC design, the basic building blocks (or "standard cells" or "macros") provided by the FPGA or ASIC manufacturer may actually be defined as 74-series MSI functions, even to the extent of having similar descriptive numbers.

that are used by digital designers to ensure that their designs are correct, manufacturable, and maintainable. Next we discuss circuit timing, a crucial element for successful digital design. Third, we describe the internal structure of combinational PLDs, which we use later as "universal" building blocks.

5.1 Documentation Standards

Good documentation is essential for correct design and efficient maintenance of digital systems. In addition to being accurate and complete, documentation must be somewhat instructive, so that a test engineer, maintenance technician, or even the original design engineer (six months after designing the circuit) can figure out how the system works just by reading the documentation.

Although the type of documentation depends on system complexity and the engineering and manufacturing environments, a documentation package should generally contain at least the following six items:

circuit specification

1. A *specification* describes exactly what the circuit or system is supposed to do, including a description of all inputs and outputs ("interfaces") and the functions that are to be performed. Note that the "spec" doesn't have to specify *how* the system achieves its results, just *what* the results are supposed to be. However, in many companies it is common practice also to incorporate one or more of the documents below into the spec to describe how the system works at the same time.

block diagram

2. A *block diagram* is an informal pictorial description of the system's major functional modules and their basic interconnections.

schematic diagram

logic diagram

3. A *schematic diagram* is a formal specification of the electrical components of the system, their interconnections, and all of the details needed to construct the system, including IC types, reference designators, and pin numbers. We've been using the term *logic diagram* for an informal drawing that does not have quite this level of detail. Most schematic drawing

DOCUMENTS ON-LINE

Professional engineering documentation nowadays is carefully maintained on corporate intranets, so it's very useful to include URLs in circuit specifications and descriptions so that references can be easily located. On-line documentation is so important and authoritative in one company that the footer on every page of every specification contains the warning that "A printed version of this document is an uncontrolled copy." That is, a printed copy could very well be obsolete.

programs have the ability to generate a *bill of materials (BOM)* from the schematic; this tells the purchasing department what electrical components they have to order to build the system.

bill of materials (BOM)

4. A *timing diagram* shows the values of various logic signals as a function of time, including the cause-and-effect delays between critical signals.

timing diagram

5. A *structured logic device description* describes the internal function of a programmable logic device (PLD), field-programmable gate array (FPGA), or application-specific integrated circuit (ASIC). It is normally written in a hardware description language (HDL) such as ABEL or VHDL, but it may be in the form of logic equations, state tables, or state diagrams. In some cases, a conventional programming language such as C may be used to model the operation of a circuit or to specify its behavior.

structured logic device description

6. A *circuit description* is a narrative text document that, in conjunction with the other documentation, explains how the circuit works internally. The circuit description should list any assumptions and potential pitfalls in the circuit's design and operation, and point out the use of any nonobvious design "tricks." A good circuit description also contains definitions of acronyms and other specialized terms, and has references to related documents.

circuit description

You've probably already seen block diagrams in many contexts. We present a few rules for drawing them in the next subsection, and then we concentrate on schematics for combinational logic circuits in the rest of this section. Section 5.2.1 introduces timing diagrams. Structured logic descriptions in the form of ABEL and VHDL programs were covered in Sections 4.6 and 4.7. In Section 11.1.6, we'll show how a C program can be used to generate the contents of a read-only memory.

The last area of documentation, the circuit description, is very important in practice. Just as an experienced programmer creates a program design document before beginning to write code, an experienced logic designer starts writing the circuit description before drawing the schematic. Unfortunately, the circuit description is sometimes the last document to be created, and sometimes it's never written at all. A circuit without a description is difficult to debug, manufacture, test, maintain, modify, and enhance.

DON'T FORGET TO WRITE!

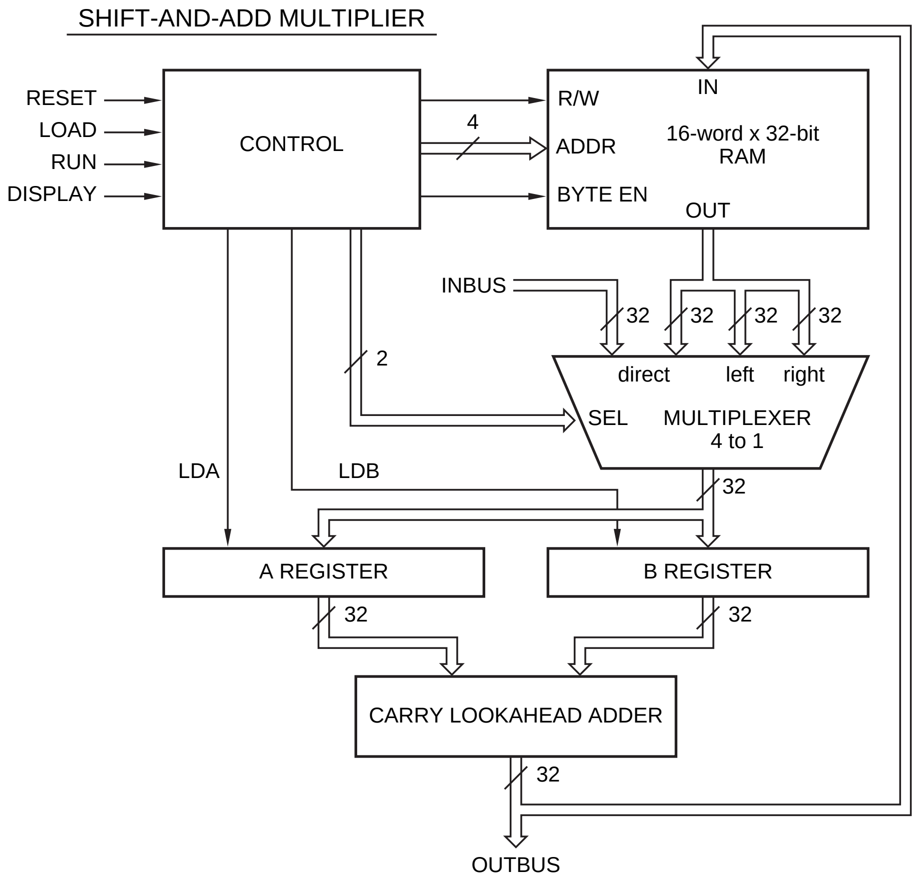
In order to create great products, logic designers must develop their language and writing skills, especially in the area of *logical* outlining and organization. The most successful logic designers (and later, project leaders, system architects, and entrepreneurs) are the ones who communicate their ideas, proposals, and decisions effectively to others. Even though it's a lot of fun to tinker in the digital design lab, don't use that as an excuse to shortchange your writing and communications courses and projects!

5.1.1 Block Diagrams

block diagram

A *block diagram* shows the inputs, outputs, functional modules, internal data paths, and important control signals of a system. In general, it should not be so detailed that it occupies more than one page, yet it must not be too vague. A small block diagram may have three to six blocks, while a large one may have 10 to 15, depending on system complexity. In any case, the block diagram must

Figure 5-1
Block diagram for a digital design project.



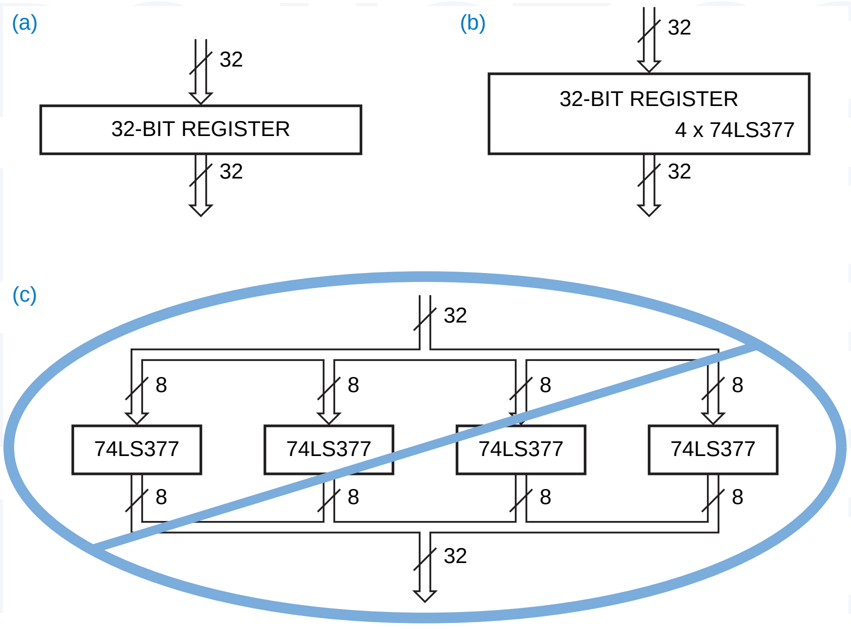


Figure 5-2
A 32-bit register block: (a) realization unspecified; (b) chips specified; (c) too much detail.

show the most important system elements and how they work together. Large systems may require additional block diagrams of individual subsystems, but there should always be a "top-level" diagram showing the entire system.

Figure 5-1 shows a sample block diagram. Each block is labeled with the function of the block, not the individual chips that comprise it. As another example, Figure 5-2(a) shows the block-diagram symbol for a 32-bit register. If the register is to be built using four 74x377 8-bit registers, and this information is important to someone reading the diagram (e.g., for cost reasons), then it can be conveyed as shown in (b). However, splitting the block to show individual chips as in (c) is incorrect.

bus

A *bus* is a collection of two or more related signal lines. In a block diagram, buses are drawn with a double or heavy line. A slash and a number may indicate how many individual signal lines are contained in a bus. Alternatively, size denoted in the bus name (e.g., INBUS[31..0] or INBUS[31:0]). Active levels (defined later) and inversion bubbles may or may not appear in block diagrams; in most cases, they are unimportant at this level of detail. However, important control signals and buses should have names, usually the same names that appear in the more detailed schematic.

The flow of control and data in a block diagram should be clearly indicated. Logic diagrams are generally drawn with signals flowing from left to right, but in block diagrams this ideal is more difficult to achieve. Inputs and outputs may be on any side of a block, and the direction of signal flow may be arbitrary. Arrowheads are used on buses and ordinary signal lines to eliminate ambiguity.

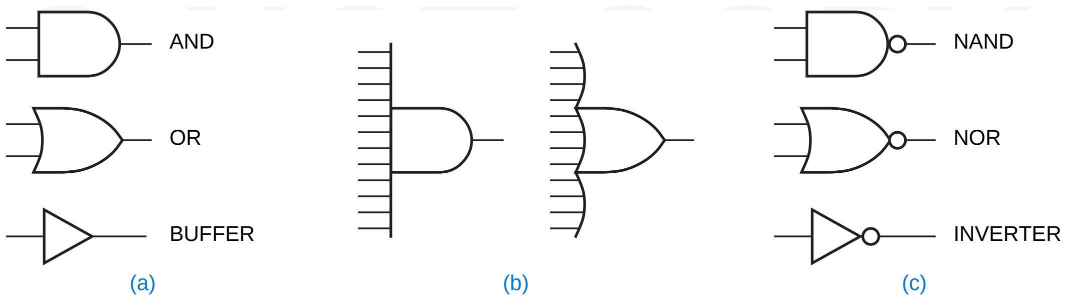


Figure 5-3 Shapes for basic logic gates: (a) AND, OR, and buffers; (b) expansion of inputs; (c) inversion bubbles.

5.1.2 Gate Symbols

The symbol shapes for AND and OR gates and buffers are shown in Figure 5-3(a). (Recall from Chapter 3 that a buffer is a circuit that simply converts "weak" logic signals into "strong" ones.) To draw logic gates with more than a few inputs, we expand the AND and OR symbols as shown in (b). A small circle, called an *inversion bubble*, denotes logical inversion or complementing and is used in the symbols for NAND and NOR gates and inverters in (c).

inversion bubble

Using the generalized DeMorgan's theorem, we can manipulate the logic expressions for gates with complemented outputs. For example, if X and Y are the inputs of a NAND gate with output Z, then we can write

$$\begin{aligned} Z &= (X \cdot Y)' \\ &= X' + Y' \end{aligned}$$

This gives rise to two different but equally correct symbols for a NAND gate, as we demonstrated in Figure 4-3 on page 199. In fact, this sort of manipulation may be applied to gates with uncomplemented inputs as well. For example, consider the following equations for an AND gate:

$$\begin{aligned} Z &= X \cdot Y \\ &= ((X \cdot Y)')' \\ &= (X' + Y')' \end{aligned}$$

IEEE STANDARD LOGIC SYMBOLS

Together with the American National Standards Institute (ANSI), the Institute of Electrical and Electronic Engineers (IEEE) has developed a standard set of logic symbols. The most recent revision of the standard is ANSI/IEEE Std 91-1984, *IEEE Standard Graphic Symbols for Logic Functions*. The standard allows both rectangular- and distinctive-shape symbols for logic gates. We have been using and will continue to use the distinctive-shape symbols throughout this book, but the rectangular-shape symbols are described in Appendix A.

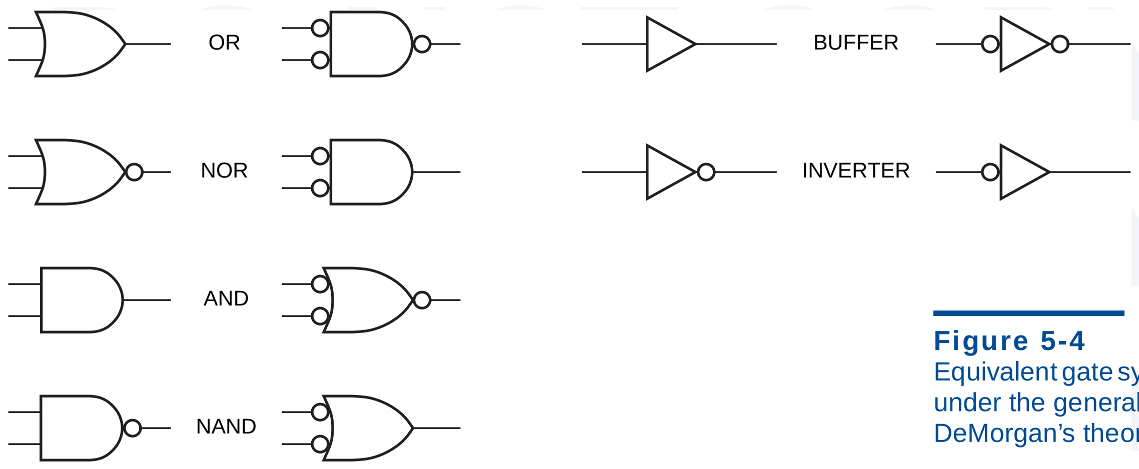


Figure 5-4
Equivalent gate symbols under the generalized DeMorgan's theorem.

Thus, an AND gate may be symbolized as an OR gate with inversion bubbles on its inputs and output.

Equivalent symbols for standard gates that can be obtained by these manipulations are summarized in Figure 5-4. Even though both symbols in a pair represent the same logic function, the choice of one symbol or the other in a logic diagram is not arbitrary, at least not if we are adhering to good documentation standards. As we'll show in the next three subsections, proper choices of signal names and gate symbols can make logic diagrams much easier to use and understand.

5.1.3 Signal Names and Active Levels

Each input and output signal in a logic circuit should have a descriptive alphanumeric label, the signal's name. Most computer-aided design systems for drawing logic circuits also allow certain special characters, such as `*`, `_`, and `!`, to be included in signal names. In the analysis and synthesis examples in Chapter 4, we used mostly single-character signal names (X, Y, etc.) because the circuits didn't do much. However, in a real system, well-chosen signal names convey information to someone reading the logic diagram the same way that variable names in a software program do. A signal's name indicates an action that is controlled (GO, PAUSE), a condition that it detects (READY, ERROR), or data that it carries (INBUS[31:0]).

active level
active high
active low
assert
negate
deassert

Each signal name should have an *active level* associated with it. A signal is *active high* if it performs the named action or denotes the named condition when it is HIGH or 1. (Under the positive-logic convention, which we use throughout this book, "HIGH" and "1" are equivalent.) A signal is *active low* if it performs the named action or denotes the named condition when it is LOW or 0. A signal is said to be *asserted* when it is at its active level. A signal is said to be *negated* (or, sometimes, *deasserted*) when it is not at its active level.

Table 5-1 Each line shows a different naming convention for active levels.

Active Low	*Active High*
READY–	READY+
ERROR.L	ERROR.H
ADDR15(L)	ADDR15(H)
RESET*	RESET
ENABLE~	ENABLE
~GO	GO
/RECEIVE	RECEIVE
TRANSMIT_L	TRANSMIT

active-level naming convention

The active level of each signal in a circuit is normally specified as part of its name, according to some convention. Examples of several different *active-level naming conventions* are shown in Table 5-1. The choice of one of these or other signal naming conventions is sometimes just a matter of personal preference, but more often it is constrained by the engineering environment. Since the active-level designation is part of the signal name, the naming convention must be compatible with the input requirements of any computer-aided design tools that will process the signal names, such as schematic editors, HDL compilers, and simulators. In this text, we'll use the last convention in the table: An active-low signal name has a suffix of _L, and an active-high signal has no suffix. The _L suffix may be read as if it were a prefix "not."

_L suffix

signal name

logic expression

logic equation

It's extremely important for you to understand the difference between signal names, expressions, and equations. A *signal name* is just a name—an alphanumeric label. A *logic expression* combines signal names using the operators of switching algebra—AND, OR, and NOT—as we explained and used throughout Chapter 4. A *logic equation* is an assignment of a logic expression to a signal name—it describes one signal's function in terms of other signals.

The distinction between signal names and logic expressions can be related to a concept used in computer programming languages: The left-hand side of an assignment statement contains a variable *name*, and the right-hand side contains an *expression* whose value will be given to the named variable (e.g., `Z = -(X+Y)` in C). In a programming language, you can't put an expression on the left-hand side of an assignment statement. In logic design, you can't use a logic expression as a signal name.

Logic signals may have names like X, READY, and GO_L. The "_L" in GO_L is just part of the signal's name, like an underscore in a variable name in a C program. There is *no* signal whose name is READY′—this is an expression, since ′ is an operator. However, there may be two signals named READY and READY_L such that READY_L = READY′ during normal operation of the circuit. We are very careful in this book to distinguish between signal names, which are always printed in black, and logic expressions, which are always printed in color when they are written near the corresponding signal lines.

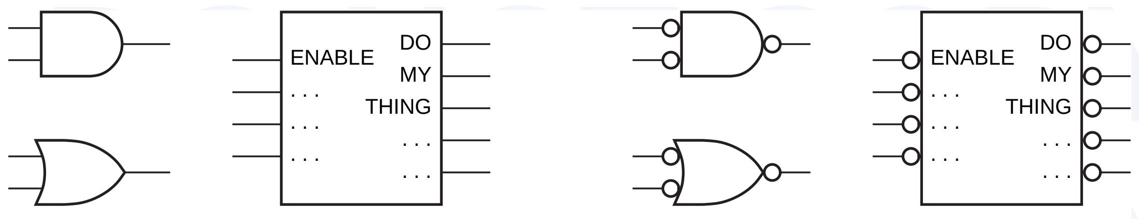


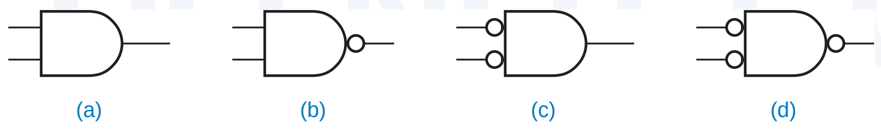
Figure 5-5 Logic symbols: (a) AND, OR, and a larger-scale logic element; (b) the same elements with active-low inputs and outputs.

5.1.4 Active Levels for Pins

When we draw the outline of an AND or OR symbol, or a rectangle representing a larger-scale logic element, we think of the given logic function as occurring *inside* that symbolic outline. In Figure 5-5(a), we show the logic symbols for an AND and OR gate and for a larger-scale element with an ENABLE input. The AND and OR gates have active-high inputs—they require 1s on the input to assert their outputs. Likewise, the larger-scale element has an active-high ENABLE input, which must be 1 to enable the element to do its thing. In (b), we show the same logic elements with active-low input and output pins. Exactly the same logic functions are performed *inside* the symbolic outlines, but the inversion bubbles indicate that 0s must now be applied to the input pins to activate the logic functions, and that the outputs are 0 when they are "doing their thing."

Thus, active levels may be associated with the input and output pins of gates and larger-scale logic elements. We use an inversion bubble to indicate an active-low pin and the absence of a bubble to indicate an active-high pin. For example, the AND gate in Figure 5-6(a) performs the logical AND of two active-high inputs and produces an active-high output: if both inputs are asserted (1), the output is asserted (1). The NAND gate in (b) also performs the AND function, but it produces an active-low output. Even a NOR or OR gate can be construed to perform the AND function using active-low inputs and outputs, as shown in (c) and (d). All four gates in the figure can be said to perform the same function: the output of each gate is asserted if both of its inputs are asserted. Figure 5-7 shows the same idea for the OR function: The output of each gate is asserted if either of its inputs is asserted.

Figure 5-6 Four ways of obtaining an AND function: (a) AND gate (74x08); (b) NAND gate (74x00); (c) NOR gate (74x02); (d) OR gate (74x32).



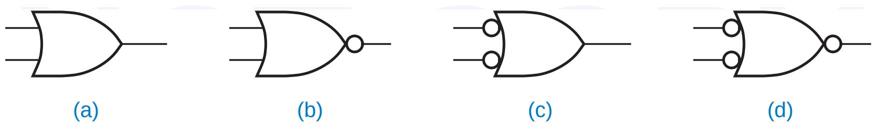


Figure 5-7 Four ways of obtaining an OR function: (a) OR gate (74x32); (b) NOR gate (74x02); (c) NAND gate (74x00); (d) AND gate (74x08).

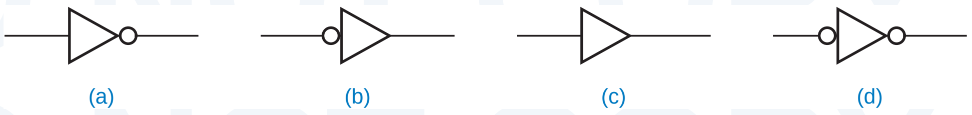


Figure 5-8 Alternate logic symbols: (a, b) inverters; (c, d) noninverting buffers.

Sometimes a noninverting buffer is used simply to boost the fanout of a logic signal without changing its function. Figure 5-8 shows the possible logic symbols for both inverters and noninverting buffers. In terms of active levels, all of the symbols perform exactly the same function: Each asserts its output signal if and only if its input is asserted.

5.1.5 Bubble-to-Bubble Logic Design

Experienced logic circuit designers formulate their circuits in terms of the logic functions performed *inside* the symbolic outlines. Whether you're designing with discrete gates or in an HDL like ABEL or VHDL, it's easiest to think of logic signals and their interactions using active-high names. However, once you're ready to realize your circuit, you may have to deal with active-low signals due to the requirements of the environment.

When you design with discrete gates, either at board or ASIC level, a key requirement is often speed. As we showed in Section 3.3.6, inverting gates are typically faster than noninverting ones, so there's often a significant performance payoff in carrying some signals in active-low form.

When you design with larger-scale elements, many of them may be off-the-shelf chips or other existing components that already have some inputs and outputs fixed in active-low form. The reasons that they use active-low signals may range from performance improvement to years of tradition, but in any case, you still have to deal with it.

NAME THAT SIGNAL!

Although it is absolutely necessary to name only a circuit's main inputs and outputs, most logic designers find it useful to name internal signals as well. During circuit debugging, it's nice to have a name to use when pointing to an internal signal that's behaving strangely. Most computer-aided design systems automatically generate labels for unnamed signals, but a user-chosen name is preferable to a computer-generated one like XSIG1057.

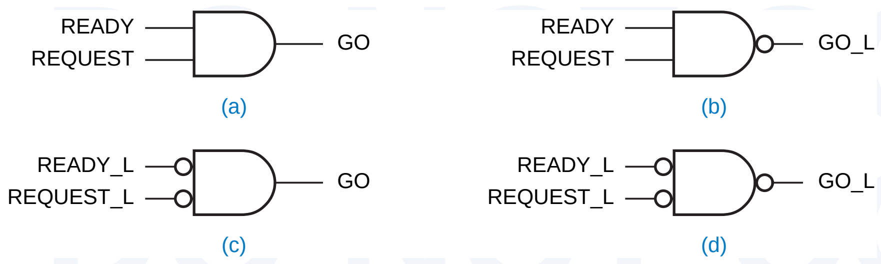


Figure 5-9 Many ways to GO: (a) active-high inputs and output; (b) active-high inputs, active-low output; (c) active-low inputs, active-high output; (d) active-low inputs and outputs.

bubble-to-bubble logic design

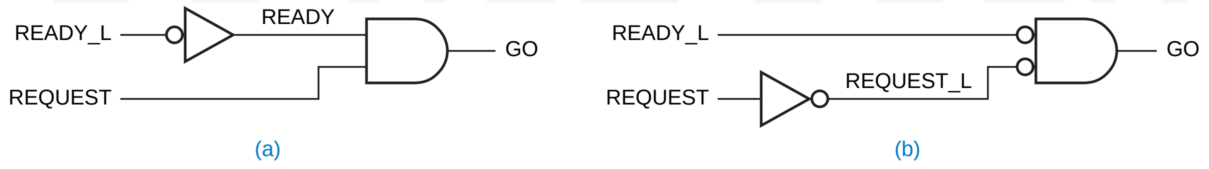
Bubble-to-bubble logic design is the practice of choosing logic symbols and signal names, including active-level designators, that make the function of a logic circuit easier to understand. Usually, this means choosing signal names and gate types and symbols so that most of the inversion bubbles “cancel out” and the logic diagram can be analyzed as if all of the signals were active high.

For example, suppose we need to produce a signal that tells a device to “GO” when we are “READY” and we get a “REQUEST.” Clearly from the problem statement, an AND function is required; in switching algebra, we would write GO = READY · REQUEST. However, we can use different gates to perform the AND function, depending on the active level required for the GO signal and the active levels of the available input signals.

Figure 5-9(a) shows the simplest case, where GO must be active-high and the available input signals are also active-high; we use an AND gate. If, on the other hand, the device that we’re controlling requires an active-low GO_L signal, we can use a NAND gate as shown in (b). If the available input signals are active-low, we can use a NOR or OR gate as shown in (c) and (d).

The active levels of available signals don’t always match the active levels of available gates. For example, suppose we are given input signals READY_L (active-low) and REQUEST (active-high). Figure 5-10 shows two different ways to generate GO using an inverter to generate the active level needed for the AND function. The second way is generally preferred, since inverting gates like NOR are generally faster than noninverting ones like AND. We drew the inverter differently in each case to make the output’s active level match its signal name.

Figure 5-10 Two more ways to GO, with mixed input levels: (a) with an AND gate; (b) with a NOR gate.



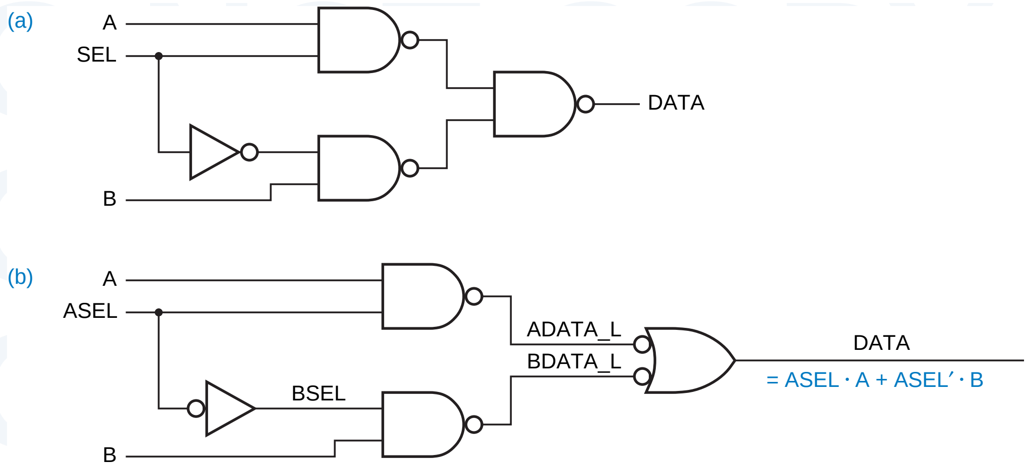
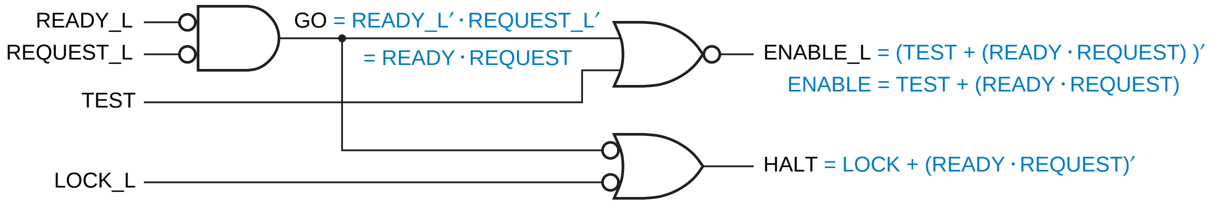


Figure 5-11 A 2-input multiplexer (you're not supposed to know what that is yet): (a) cryptic logic diagram; (b) proper logic diagram using active-level designators and alternate logic symbols.

To understand the benefits of bubble-to-bubble logic design, consider the circuit in Figure 5-11(a). What does it do? In Section 4.2 we showed several ways to analyze such a circuit, and we could certainly obtain a logic expression for the DATA output using these techniques. However, when the circuit is redrawn in Figure 5-11(b), the output function can be read directly from the logic diagram, as follows. The DATA output is asserted when either ADATA_L or BDATA_L is asserted. If ASEL is asserted, then ADATA_L is asserted if and only if A is asserted; that is, ADATA_L is a copy of A. If ASEL is negated, BSEL is asserted and BDATA_L is a copy of B. In other words, DATA is a copy of A if ASEL is asserted, and DATA is a copy of B if ASEL is negated. Even though there are five inversion bubbles in the logic diagram, we mentally had to perform only one negation to understand the circuit—that BSEL is asserted if ASEL is not asserted.

If we wish, we can write an algebraic expression for the DATA output. We use the technique of Section 4.2, simply propagating expressions through gates toward the output. In doing so, we can ignore pairs of inversion bubbles that cancel, and directly write the expression shown in color in the figure.

Figure 5-12 Another properly drawn logic diagram.



READY_L
REQUEST_L
TEST
LOCK_L
GO = READY_L′ · REQUEST_L′
= READY · REQUEST
ENABLE_L = (TEST + (READY · REQUEST))′
ENABLE = TEST + (READY · REQUEST)
HALT = LOCK + (READY · REQUEST)′

BUBBLE-TO-BUBBLE LOGIC DESIGN RULES

The following rules are useful for performing bubble-to-bubble logic design:

- The signal name on a device's output should have the same active level as the device's output pin, that is, active-low if the device symbol has an inversion bubble on the output pin, active-high if not.
- If the active level of an input signal is the same as that of the input pin to which it is connected, then the logic function inside the symbolic outline is activated when the signal is asserted. This is the most common case in a logic diagram.
- If the active level of an input signal is the opposite of that of the input pin to which it is connected, then the logic function inside the symbolic outline is activated when the signal is negated. This case should be avoided whenever possible because it forces us to keep track mentally of a logical negation to understand the circuit.

Another example is shown in Figure 5-12. Reading directly from the logic diagram, we see that ENABLE_L is asserted if READY_L and REQUEST_L are asserted or if TEST is asserted. The HALT output is asserted if READY_L and REQUEST_L are not both asserted or if LOCK_L is asserted. Once again, this example has only one place where a gate input's active level does not match the input signal level, and this is reflected in the verbal description of the circuit.

We can, if we wish, write algebraic equations for the ENABLE_L and HALT outputs. As we propagate expressions through gates towards the output, we obtain expressions like $\text{READY\_L}' \cdot \text{REQUEST}'$. However, we can use our active-level naming convention to simplify terms like READY_L′. The circuit contains no signal with the name READY; but if it did, it would satisfy the relationship $\text{READY} = \text{READY\_L}'$ according to the naming convention. This allows us to write the ENABLE_L and HALT equations as shown. Complementing both sides of the ENABLE_L equation, we obtain an equation that describes a hypothetical active-high ENABLE output in terms of hypothetical active-high inputs.

We'll see more examples of bubble-to-bubble logic design in this and later chapters, especially as we begin to use larger-scale logic elements.

5.1.6 Drawing Layout

Logic diagrams and schematics should be drawn with gates in their "normal" orientation with inputs on the left and outputs on the right. The logic symbols for larger-scale logic elements are also normally drawn with inputs on the left and outputs on the right.

A complete schematic page should be drawn with system inputs on the left and outputs on the right, and the general flow of signals should be from left to right. If an input or output appears in the middle of a page, it should be extended to the left or right edge, respectively. In this way, a reader can find all inputs and outputs by looking at the edges of the page only. All signal paths on the page

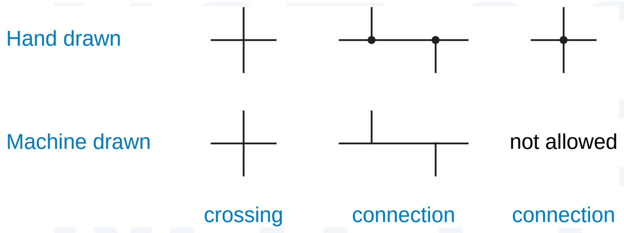


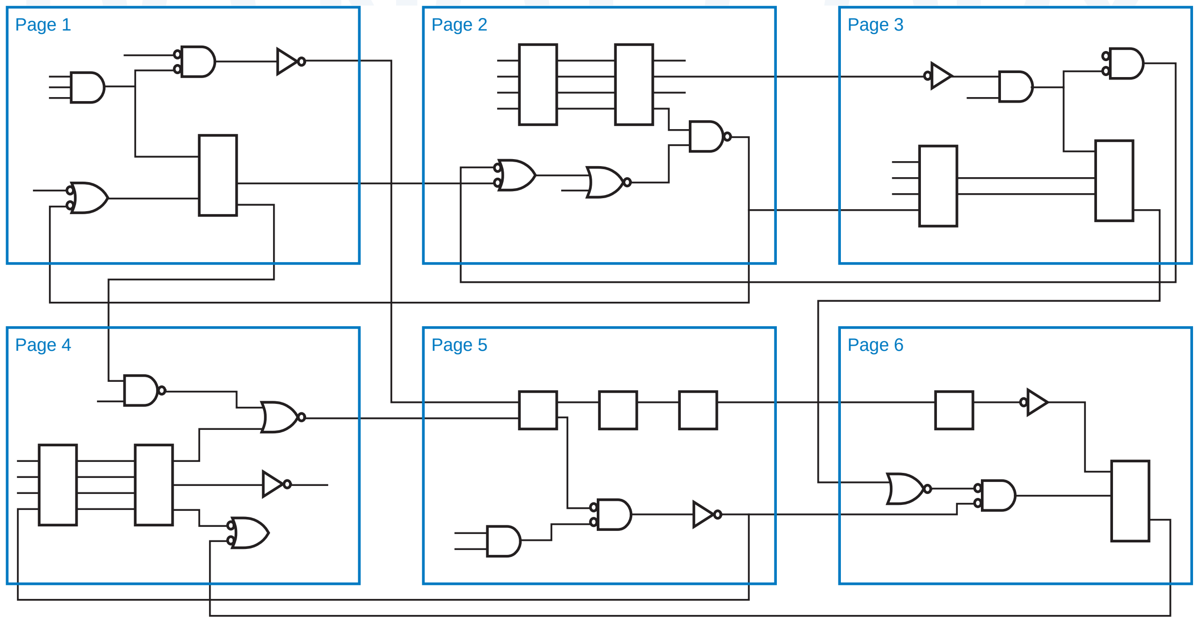
Figure 5-13 Line crossings and connections.

should be connected when possible; paths may be broken if the drawing gets crowded, but breaks should be flagged in both directions, as described later.

Sometimes block diagrams are drawn without crossing lines for a neater appearance, but this is never done in logic diagrams. Instead, lines are allowed to cross and connections are indicated clearly with a dot. Still, some computer-aided design systems (and some designers) can't draw legible connection dots. To distinguish between crossing lines and connected lines, they adopt the convention that only "T"-type connections are allowed, as shown in Figure 5-13. This is a good convention to follow in any case.

Schematics that fit on a single page are the easiest to work with. The largest practical paper size for a schematic might be E-size (34"×44"). Although its drawing capacity is great, such a large paper size is unwieldy to work with. The best compromise of drawing capacity and practicality is B-size (11"×17"). It can be easily folded for storage and quick reference in standard 3-ring notebooks, and it can be copied on most office copiers. Regardless of paper size, schematics

Figure 5-14 Flat schematic structure.



come out best when the page is used in landscape format, that is, with its long dimension oriented from left to right, the direction of most signal flow.

Schematics that don't fit on a single page should be broken up into individual pages in a way that minimizes the connections (and confusion) between pages. They may also use a coordinate system, like that of a road map, to flag the sources and destinations of signals that travel from one page to another. An outgoing signal should have flags referring to all of the destinations of that signal, while an incoming signal should have a flag referring to the source only. That is, an incoming signal should be flagged to the place where it is generated, not to a place somewhere in the middle of a chain of destinations that use the signal.

signal flags

A multiple-page schematic usually has a "flat" structure. As shown in Figure 5-14, each page is carved out from the complete schematic and can connect to any other page as if all the pages were on one large sheet. However, much like programs, schematics can also be constructed hierarchically, as illustrated in Figure 5-15. In this approach, the "top-level" schematic is just a single page that may take the place of a block diagram. Typically, the top-level schematic con-

flat schematic structure

hierarchical schematic structure

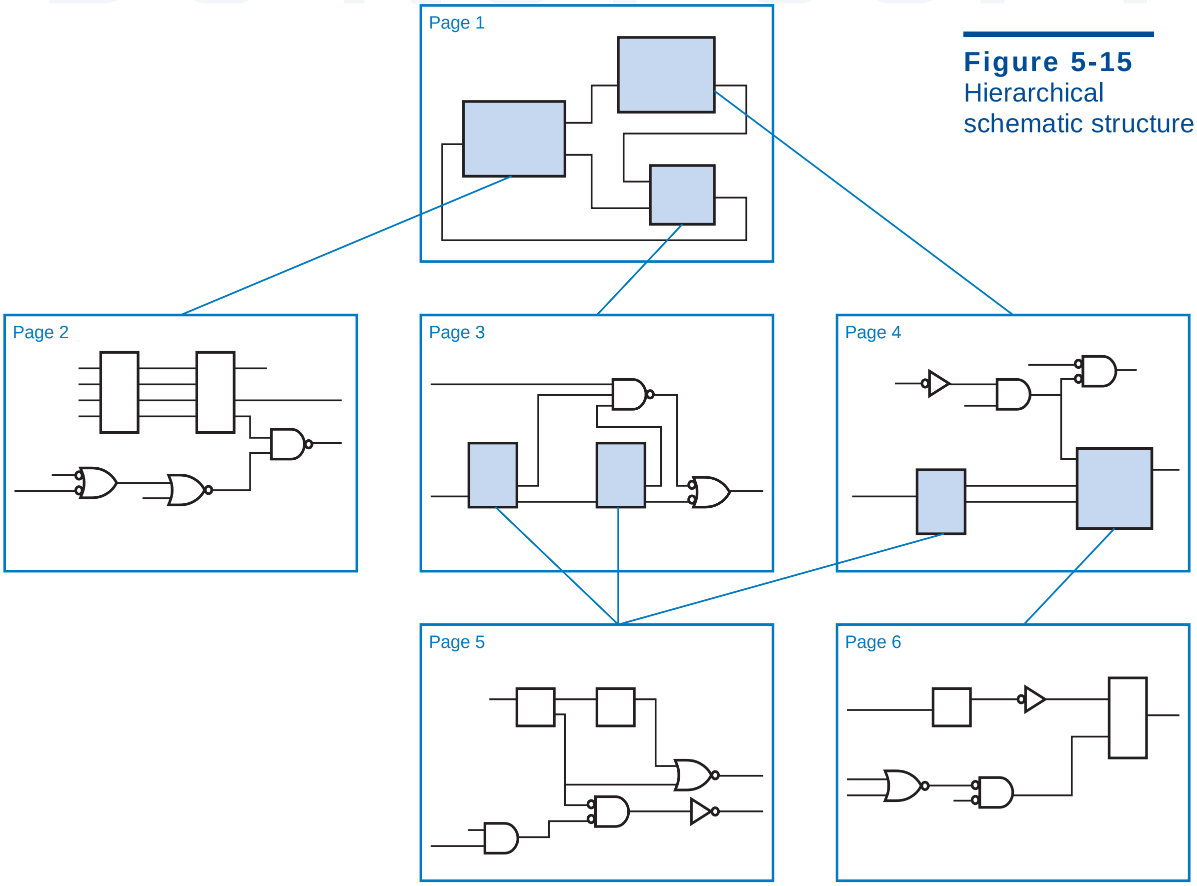


Figure 5-15 Hierarchical schematic structure.

tains no gates or other logic elements; it only shows blocks corresponding to the major subsystems, and their interconnections. The blocks or subsystems are turn on lower-level pages, which may contain ordinary gate-level descriptions, or may themselves use blocks defined in lower-level hierarchies. If a particular lower-level hierarchy needs to be used more than once, it may be reused (or “called,” in the programming sense) multiple times by the higher-level pages.

Most computer-aided logic design systems support both flat and hierarchical schematics. Proper signal naming is very important in both styles, since there are a number of common errors that can occur:

- Like any other program, a schematic-entry program does what you say, not what you mean. If you use slightly different names for what you intend to be the same signal on different pages, they won’t be connected.
- Conversely, if you inadvertently use the same name for different signals on different pages of a flat schematic, many programs will dutifully connect them together, even if you haven’t connected them with an off-page flag. (In a hierarchical schematic, reusing a name at different places in the hierarchy is generally OK, because the program qualifies each name with its position in the hierarchy.)
- In a hierarchical schematic, you have to be careful in naming the external interface signals on pages in the lower levels of the hierarchy. These are the names that will appear inside the blocks corresponding to these pages when they are used at higher levels of the hierarchy. It’s very easy to transpose signal names or use a name with the wrong active level, yielding incorrect results when the block is used.
- This is not usually a naming problem, but all schematic programs seem to have quirks in which signals that appear to be connected are not. Using the “T” convention in Figure 5-13 can help minimize this problem.

Fortunately, most schematic programs have error-checking facilities that can catch many of these errors, for example, by searching for signal names that have no inputs, no outputs, or multiple outputs associated with them. But most logic designers learn the importance of careful, manual schematic double-checking only through the bitter experience of building a printed-circuit board or an ASIC based on a schematic containing some dumb error.

5.1.7 Buses

As defined previously, a bus is a collection of two or more related signal lines. For example, a microprocessor system might have an address bus with 16 lines, ADDR0–ADDR15, and a data bus with 8 lines, DATA0–DATA7. The signal names in a bus are not necessarily related or ordered as in these first examples. For example, a microprocessor system might have a control bus containing five signals, ALE, MIO, RD_L, WR_L, and RDY.

Logic diagrams use special notation for buses in order to reduce the amount of drawing and to improve readability. As shown in Figure 5-16, a bus has its own descriptive name, such as ADDR[15:0], DATA[7:0], or CONTROL. A bus name may use brackets and a colon to denote a range. Buses are drawn with thicker lines than ordinary signals. Individual signals are put into or pulled out of the bus by connecting an ordinary signal line to the bus and writing the signal name. Often a special connection dot is also used, as in the example.

A computer-aided design system keeps track of the individual signals in a bus. When it actually comes time to build a circuit from the schematic, signal lines in a bus are treated just as though they had all been drawn individually.

The symbols at the right-hand edge of Figure 5-16 are interpage signal flags. They indicate that LA goes out to page 2, DB is bidirectional and connects to page 2, and CONTROL is bidirectional and connects to pages 2 and 3.

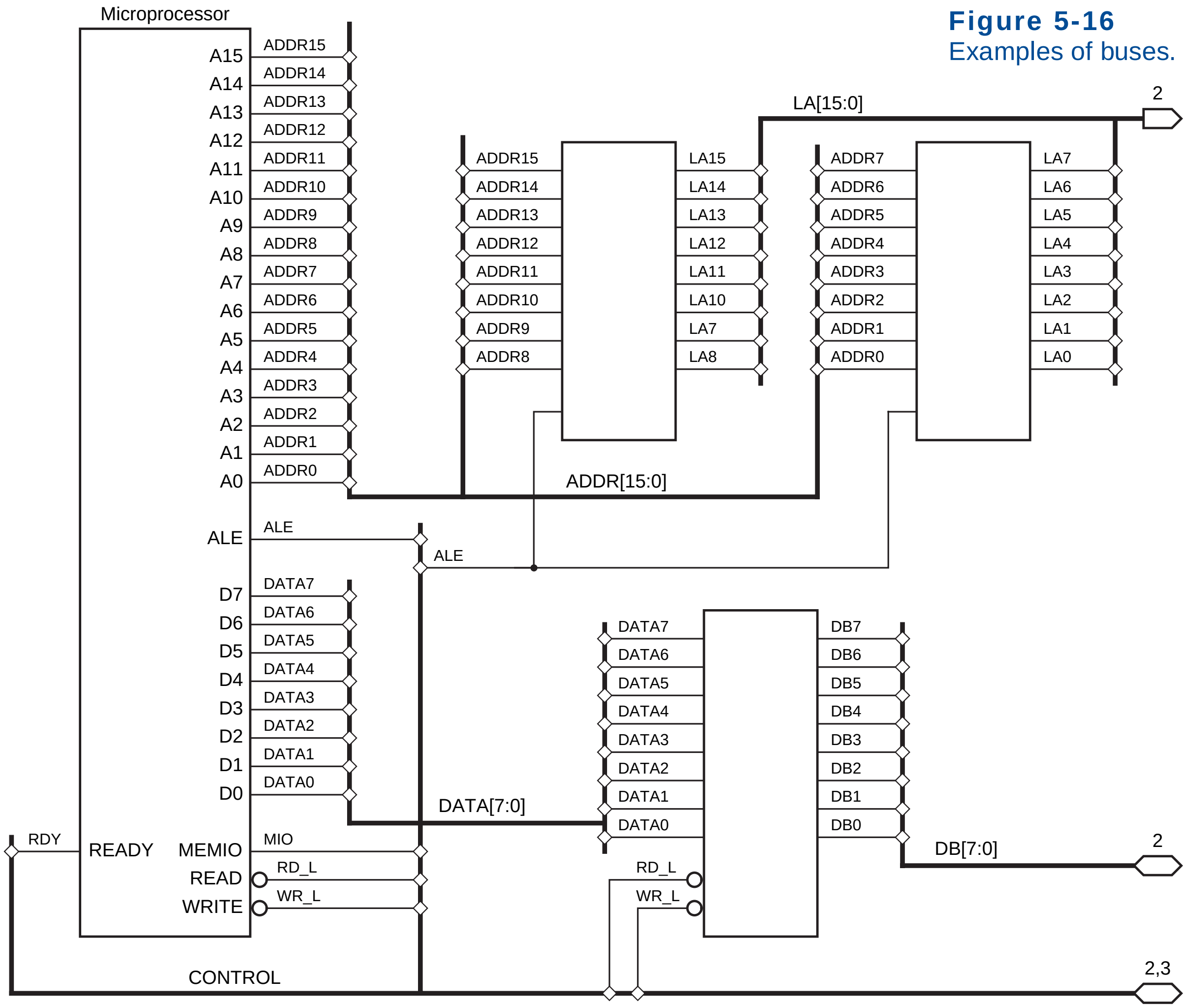

Microprocessor
A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
ADDR15 ADDR14 ADDR13 ADDR12 ADDR11 ADDR10 ADDR9 ADDR8 ADDR7 ADDR6 ADDR5 ADDR4 ADDR3 ADDR2 ADDR1 ADDR0
LA15 LA14 LA13 LA12 LA11 LA10 LA7 LA8
LA7 LA6 LA5 LA4 LA3 LA2 LA1 LA0
LA[15:0]
2
ADDR[15:0]
ALE
D7 D6 D5 D4 D3 D2 D1 D0
DATA7 DATA6 DATA5 DATA4 DATA3 DATA2 DATA1 DATA0
DATA[7:0]
DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
DB[7:0]
RDY
READY
MEMIO
READ
WRITE
MIO
RD_L
WR_L
CONTROL
2,3

Figure 5-16
Examples of buses.

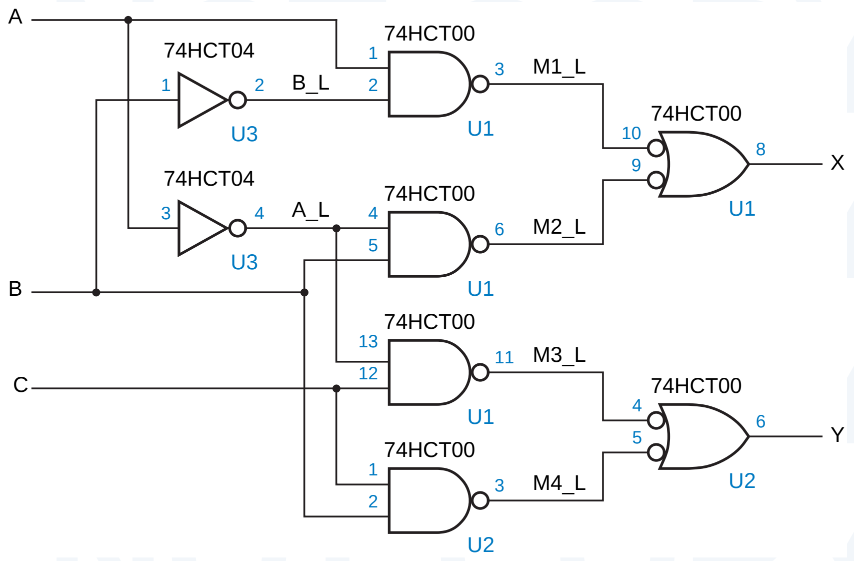


Figure 5-17
Schematic diagram for a circuit using a 74HCT00.

5.1.8 Additional Schematic Information

Complete schematic diagrams indicate IC types, reference designators, and pin numbers, as in Figure 5-17. The *IC type* is a part number identifying the integrated circuit that performs a given logic function. For example, a 2-input NAND gate might be identified as a 74HCT00 or a 74LS00. In addition to the logic function, the IC type identifies the device's logic family and speed.

IC type

The *reference designator* for an IC identifies a particular instance of that IC type installed in the system. In conjunction with the system's mechanical documentation, the reference designator allows a particular IC to be located during assembly, test, and maintenance of the system. Traditionally, reference designators for ICs begin with the letter U (for "unit").

reference designator

Once a particular IC is located, *pin numbers* are used to locate individual logic signals on its pins. The pin numbers are written near the corresponding inputs and outputs of the standard logic symbol, as shown in Figure 5-17.

pin number

In the rest of this book, just to make you comfortable with properly drawn schematics, we'll include reference designators and pin numbers for all of the logic circuit examples that use SSI and MSI parts.

Figure 5-18 shows the pinouts of many different SSI ICs that are used in examples throughout this book. Some special graphic elements appear in a few of the symbols:

- Symbols for the 74x14 Schmitt-trigger inverter has a special element inside the symbol to indicate hysteresis.
- Symbols for the 74x03 quad NAND and the 74x266 quad Exclusive NOR have a special element to indicate an open-drain or open-collector output.

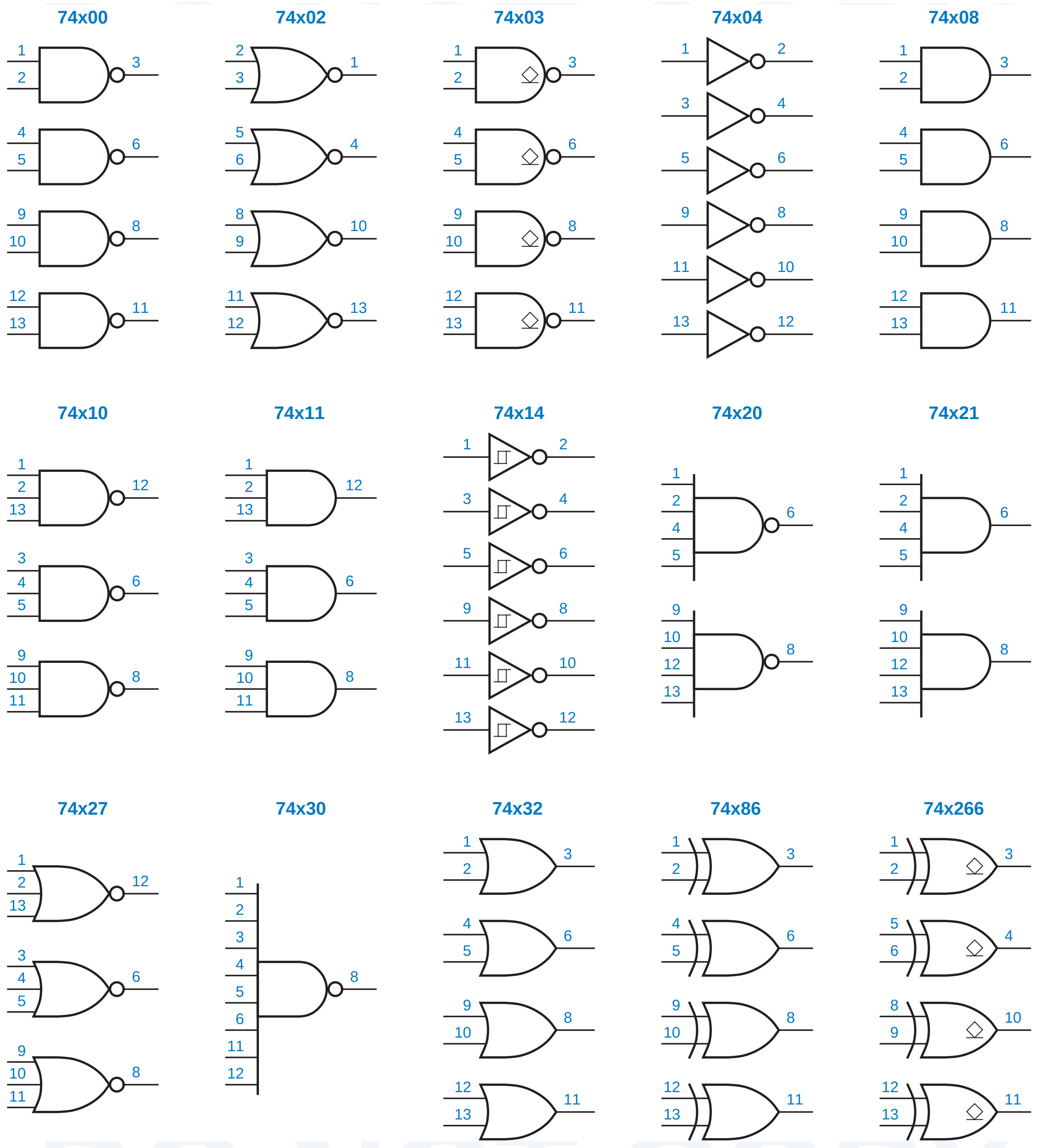


Figure 5-18 Pinouts for SSI ICs in standard dual-inline packages.

When you prepare a schematic diagram for a board-level design using a schematic drawing program, the program automatically provides the pin numbers for the devices that you select from its component library. Note that an IC's pin numbers may differ depending on package type, so you have to be careful to select the right version of the component from the library. Figure 5-18 shows the pin numbers that are used in a dual-inline package, the type of package that you would use in a digital design laboratory course or in a low-density, "thru-hole" commercial printed-circuit board.

5.2 Circuit Timing

"Timing is everything"—in investing, in comedy, and yes, in digital design. As we studied in Section 3.6, the outputs of real circuits take time to react to their inputs, and many of today's circuits and systems are so fast that even the speed-of-light delay in propagating an output signal to an input on the other side of a board or chip is significant.

Most digital systems are sequential circuits that operate step-by-step under the control of a periodic clock signal, and the speed of the clock is limited by the worst-case time that it takes for the operations in one step to complete. Thus, digital designers need to be keenly aware of timing behavior in order to build fast circuits that operate correctly under all conditions.

The last several years have seen great advances in the number and quality of CAD tools for analyzing circuit timing. Still, quite often the greatest challenge in completing a board-level or especially an ASIC design is achieving the required timing performance. In this section, we start with the basics, so you can understand what the tools are doing when you use them, and so you can figure out how to fix your circuits when their timing isn't quite making it.

5.2.1 Timing Diagrams

timing diagram

A *timing diagram* illustrates the logical behavior of signals in a digital circuit as a function of time. Timing diagrams are an important part of the documentation of any digital system. They can be used both to explain the timing relationships among signals within a system, and to define the timing requirements of external signals that are applied to the system.

Figure 5-19(a) is the block diagram of a simple combinational circuit with two inputs and two outputs. Assuming that the ENB input is held at a constant value, (b) shows the delay of the two outputs with respect to the GO input. In each waveform, the upper line represents a logic 1, and the lower line a logic 0. Signal transitions are drawn as slanted lines to remind us that they do not occur in zero time in real circuits. (Also, slanted lines look nicer than vertical ones.)

causality

delay

Arrows are sometimes drawn, especially in complex timing diagrams, to show *causality*—which input transitions cause which output transitions. In any case, the most important information provided by a timing diagram is a specification of the *delay* between transitions.

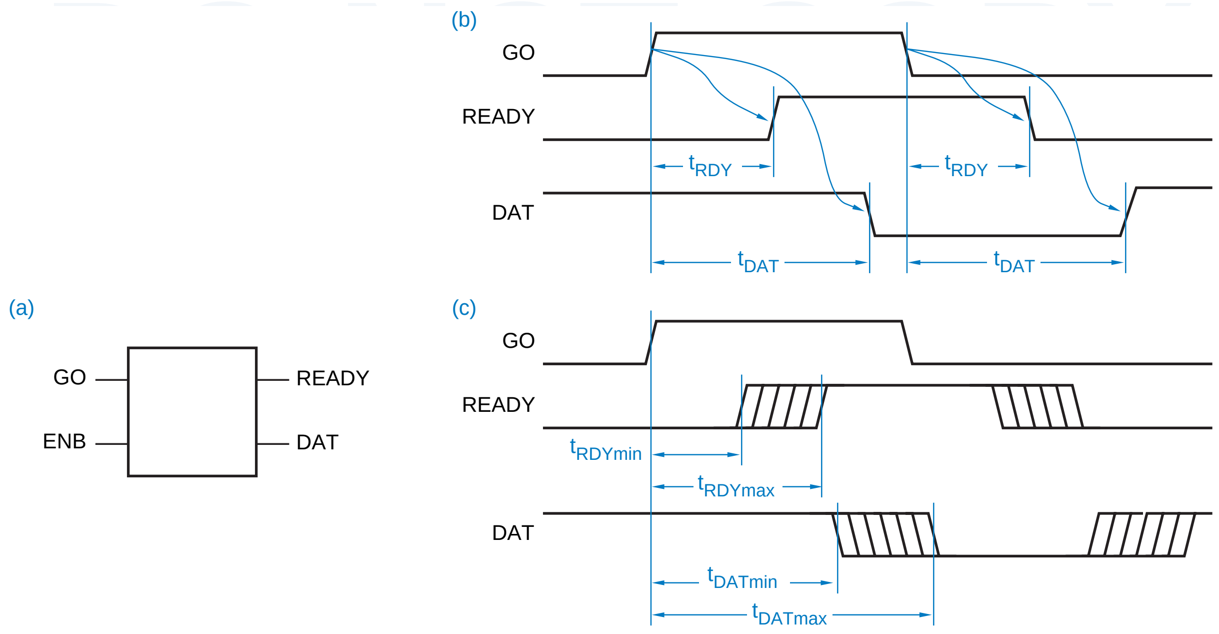


Figure 5-19 Timing diagrams for a combinational circuit: (a) block diagram of circuit; (b) causality and propagation delay; (c) minimum and maximum delays.

Different paths through a circuit may have different delays. For example, Figure 5-19(b) shows that the delay from GO to READY is shorter than the delay from GO to DAT. Similarly, the delays from the ENB input to the outputs may vary, and could be shown in another timing diagram. And, as we'll discuss later, the delay through any given path may vary depending on whether the output is changing from LOW to HIGH or from HIGH to LOW (this phenomenon is not shown in the figure).

Delay in a real circuit is normally measured between the centerpoints of transitions, so the delays in a timing diagram are marked this way. A single timing diagram may contain many different delay specifications. Each different delay is marked with a different identifier, such as t_{RDY} and t_{DAT} in the figure. In large timing diagrams, the delay identifiers are usually numbered for easier reference (e.g., t_1, t_2, …, t_{42}). In either case, the timing diagram is normally accompanied by a *timing table* that specifies each delay amount and the conditions under which it applies.

timing table

Since the delays of real digital components can vary depending on voltage, temperature, and manufacturing parameters, delay is seldom specified as a single number. Instead, a timing table may specify a range of values by giving *minimum*, *typical*, and *maximum* values for each delay. The idea of a range of delays is sometimes carried over into the timing diagram itself by showing the transitions to occur at uncertain times, as in Figure 5-19(c).

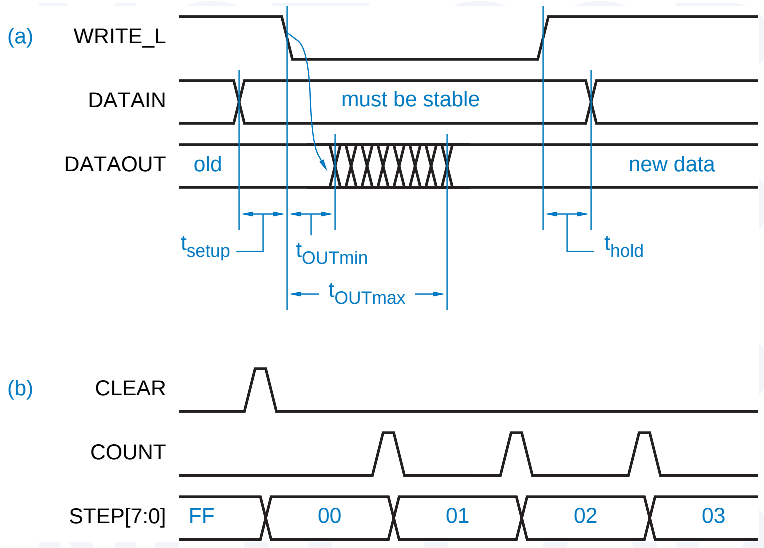


Figure 5-20 Timing diagrams for "data" signals: (a) certain and uncertain transitions; (b) sequence of values on an 8-bit bus.

For some signals, the timing diagram needn't show whether the signal changes from 1 to 0 or from 0 to 1 at a particular time, only that a transition occurs then. Any signal that carries a bit of "data" has this characteristic—the actual value of the data bit varies according to circumstances but, regardless of value, the bit is transferred, stored, or processed at a particular time relative to "control" signals in the system. Figure 5-20(a) is a timing diagram that illustrates this concept. The "data" signal is normally at a steady 0 or 1 value, and transitions occur only at the times indicated. The idea of an uncertain delay time can also be used with "data" signals, as shown for the DATAOUT signal.

Quite often in digital systems, a group of data signals in a bus is processed by identical circuits. In this case, all signals in the bus have the same timing, and can be represented by a single line in the timing diagram and corresponding specifications in the timing table. If the bus bits are known to take on a particular combination at a particular time, this is sometimes shown in the timing diagram using binary, octal, or hexadecimal numbers, as in Figure 5-20(b).

5.2.2 Propagation Delay

propagation delay

In Section 3.6.2, we formally defined the *propagation delay* of a signal path as the time that it takes for a change at the input of the path to produce a change at the output of the path. A combinational circuit with many inputs and outputs has many different paths, and each one may have a different propagation delay. Also, the propagation delay when the output changes from LOW to HIGH (t_{pLH}) may be different from the delay when it changes from HIGH to LOW (t_{pHL}).

The manufacturer of a combinational-logic IC normally specifies all of these different propagation delays, or at least the delays that would be of interest in typical applications. A logic designer who combines ICs in a larger circuit uses the individual device specifications to analyze the overall circuit timing. The delay of a path through the overall circuit is the sum of the delays through subpaths in the individual devices.

5.2.3 Timing Specifications

The timing specification for a device may give minimum, typical, and maximum values for each propagation-delay path and transition direction:

- *Maximum*. This specification is the one that is most often used by experienced designers, since a path "never" has a propagation delay longer than the maximum. However, the definition of "never" varies among logic families and manufacturers. For example, "maximum" propagation delays of 74LS and 74S TTL devices are specified with $V_{CC} = 5$ V, $T_A = 25$ºC, and almost no capacitive load. If the voltage or temperature is different, or if the capacitive load is more than 15 pF, the delay may be longer. On the other hand, a "maximum" propagation delay is specified for 74AC and 74ACT devices over the full operating voltage and temperature range, and with a heavier capacitive load of 50 pF.

maximum delay

- *Typical*. This specification is the one that is most often used by designers who don't expect to be around when their product leaves the friendly environment of the engineering lab and is shipped to customers. The "typical" delay is what you see from a device that was manufactured on a good day and is operating under near-ideal conditions.

typical delay

- *Minimum*. This is the smallest propagation delay that a path will ever exhibit. Most well-designed circuits don't depend on this number; that is, they will work properly even if the delay is zero. That's good because manufacturers don't specify minimum delay in most moderate-speed logic families, including 74LS and 74S TTL. However, in high-speed families, including ECL and 74AC and 74ACT CMOS, a nonzero minimum delay is specified to help the designer ensure that timing requirements of latches and flip-flops discussed in \secref{llff}, are met.

minimum delay

Table 5-2 lists the typical and maximum delays of several 74-series CMOS and TTL gates. Table 5-3 does the same thing for most of the CMOS and TTL MSI parts that are introduced later in this chapter.

HOW TYPICAL IS TYPICAL?

Most ICs, perhaps 99%, really are manufactured on "good" days and exhibit delays near the "typical" specifications. However, if you design a system that works only if all of its 100 ICs meet the "typical" timing specs, probability theory suggests that 63% $(1 - .99^{100})$ of the systems won't work.But see the next box....

Table 5-2 Propagation delay in nanoseconds of selected 5-V CMOS and TTL SSI parts.

	74HCT		*74AHCT*				*74LS*			
	Typical	*Maximum*	*Typical*		*Maximum*		*Typical*		*Maximum*	
Part number	$t_{\mathrm{pLH}}, t_{\mathrm{pHL}}$	$t_{\mathrm{pLH}}, t_{\mathrm{pHL}}$	t_{pLH}	t_{pHL}	t_{pLH}	t_{pHL}	t_{pLH}	t_{pHL}	t_{pLH}	t_{pHL}
’00, ’10	11	35	5.5	5.5	9.0	9.0	9	10	15	15
’02	9	29	3.4	4.5	8.5	8.5	10	10	15	15
‘04	11	35	5.5	5.5	8.5	8.5	9	10	15	15
’08, ’11	11	35	5.5	5.5	9.0	9.0	8	10	15	20
’14	16	48	5.5	5.5	9.0	9.0	15	15	22	22
’20	11	35					9	10	15	15
’21	11	35					8	10	15	20
’27	9	29	5.6	5.6	9.0	9.0	10	10	15	15
’30	11	35					8	13	15	20
’32	9	30	5.3	5.3	8.5	8.5	14	14	22	22
’86 (2 levels)	13	40	5.5	5.5	10	10	12	10	23	17
’86 (3 levels)	13	40	5.5	5.5	10	10	20	13	30	22

A COROLLARY OF MURPHY’S LAW

Murphy’s law states, “If something can go wrong, it will.” A corollary to this is, “If you want something to go wrong, it won’t.”

In the boxed example on the previous page, you might think that you have a 63% chance of detecting the potential timing problems in the engineering lab. The problems aren’t spread out evenly, though, since all ICs from a given batch tend to behave about the same. Murphy’s Corollary says that *all* of the engineering prototypes will be built with ICs from the same, “good” batches. Therefore, everything works fine for a while, just long enough for the system to get into volume production and for everyone to become complacent and self-congratulatory.

Then, unbeknownst to the production department, a “slow” batch of some IC type arrives from a supplier and gets used in every system that is built, so that *nothing* works. The production engineers scurry around trying to analyze the problem (not easy, because the designer is long gone and didn’t bother to write a circuit description), and in the meantime the company loses big bucks because it is unable to ship its product.

Table 5-3 Propagation delay in nanoseconds of selected CMOS and TTL MSI parts.

			74HCT		*74AHCT / FCT*		*74LS*			
			Typical	*Maximum*	*Typical*	*Maximum*	*Typical*		*Maximum*	
Part	*From*	*To*	t_{pLH}, t_{pHL}	t_{pLH}, t_{pHL}	t_{pLH}, t_{pHL}	t_{pLH}, t_{pHL}	t_{pLH}	t_{pHL}	t_{pLH}	t_{pHL}
’138	any select	output (2)	23	45	8.1 / 5	13 / 9	11	18	20	41
	any select	output (3)	23	45	8.1 / 5	13 / 9	21	20	27	39
	$\overline{\text{G2A}}$, $\overline{\text{G2B}}$	output	22	42	7.5 / 4	12 / 8	12	20	18	32
	G1	output	22	42	7.1 / 4	11.5 / 8	14	13	26	38
’139	any select	output (2)	14	43	6.5 / 5	10.5 / 9	13	22	20	33
	any select	output (3)	14	43	6.5 / 5	10.5 / 9	18	25	29	38
	enable	output	11	43	5.9 / 5	9.5 / 9	16	21	24	32
’151	any select	Y	17	51	- / 5	- / 9	27	18	43	30
	any select	$\overline{\text{Y}}$	18	54	- / 5	- / 9	14	20	23	32
	any data	Y	16	48	- / 4	- / 7	20	16	32	26
	any data	$\overline{\text{Y}}$	15	45	- / 4	- / 7	13	12	21	20
	enable	Y	12	36	- / 4	- / 7	26	20	42	32
	enable	$\overline{\text{Y}}$	15	45	- / 4	- / 7	15	18	24	30
’153	any select	output	14	43	- / 5	- / 9	19	25	29	38
	any data	output	12	43	- / 4	- / 7	10	17	15	26
	enable	output	11	34	- / 4	- / 7	16	21	24	32
’157	select	output	15	46	6.8 / 7	11.5 / 10.5	15	18	23	27
	any data	output	12	38	5.6 / 4	9.5 / 6	9	9	14	14
	enable	output	12	38	7.1 / 7	12.0 / 10.5	13	14	21	23
’182	any $\overline{\text{Gi}}$, $\overline{\text{Pi}}$	C1–3	13	41			4.5	4.5	7	7
	any $\overline{\text{Gi}}$, $\overline{\text{Pi}}$	$\overline{\text{G}}$	13	41			5	7	7.5	10.5
	any $\overline{\text{Pi}}$	$\overline{\text{P}}$	11	35			4.5	6.5	6.5	10
	C0	C1–3	17	50			6.5	7	10	10.5
’280	any input	EVEN	18	53	- / 6	- / 10	33	29	50	45
	any input	ODD	19	56	- / 6	- / 10	23	31	35	50
’283	C0	any Si	22	66			16	15	24	24
	any Ai, Bi	any Si	21	61			15	15	24	24
	C0	C4	19	58			11	11	17	22
	any Ai, Bi	C4	20	60			11	12	17	17
’381	CIN	any Fi					18	14	27	21
	any Ai, Bi	$\overline{\text{G}}$					20	21	30	33
	any Ai, Bi	$\overline{\text{P}}$					21	33	23	33
	any Ai, Bi	any Fi					20	15	30	23
	any select	any Fi					35	34	53	51
	any select	$\overline{\text{G}}$, $\overline{\text{P}}$					31	32	47	48
’682	any Pi	$\overline{\text{PEQQ}}$	26	69	- / 7	- / 11	13	15	25	25
	any Qi	$\overline{\text{PEQQ}}$	26	69	- / 7	- / 11	14	15	25	25
	any Pi	$\overline{\text{PGTQ}}$	26	69	- / 9	- / 14	20	15	30	30
	any Qi	$\overline{\text{PGTQ}}$	26	69	- / 9	- / 14	21	19	30	30

ESTIMATING MINIMUM DELAYS

If the minimum delay of an IC is not specified, a conservative designer assumes that it has a minimum delay of zero.

Some circuits won't work if the propagation delay actually goes to zero, but the cost of modifying a circuit to handle the zero-delay case may be unreasonable, especially since this case is expected never to occur. To obtain a design that always works under "reasonable" conditions, logic designers often estimate that ICs have minimum delays of one-fourth to one-third of their published *typical* delays.

All inputs of an SSI gate have the same propagation delay to the output. Note that TTL gates usually have different delays for LOW-to-HIGH and HIGH-to-LOW transitions (t_{pLH} and t_{pHL}), but CMOS gates usually do not. CMOS gates have a more symmetrical output driving capability, so any difference between the two cases is usually not worth noting.

The delay from an input transition to the corresponding output transition depends on the internal path taken by the changing signal, and in larger circuits the path may be different for different input combinations. For example, the 74LS86 2-input XOR gate is constructed from four NAND gates as shown in Figure 5-70 on page 372, and has two different-length paths from either input to the output. If one input is LOW, and the other is changed, the change propagates through two NAND gates, and we observe the first set of delays shown in Table 5-2. If one input is HIGH, and the other is changed, the change propagates through *three* NAND gates internally, and we observe the second set of delays. Similar behavior is exhibited by the 74LS138 and 74LS139 in Table 5-3. However, the corresponding CMOS parts do not show these differences; they are small enough to be ignored.

5.2.4 Timing Analysis

To accurately analyze the timing of a circuit containing multiple SSI and MSI devices, a designer may have to study its logical behavior in excruciating detail. For example, when TTL inverting gates (NAND, NOR, etc.) are placed in series, a LOW-to-HIGH change at one gate's output causes a HIGH-to-LOW change at the next one's, and so the differences between t_{pLH} and t_{pHL} tend to average out. On the other hand, when noninverting gates (AND, OR, etc.) are placed in series, a transition causes all outputs to change in the same direction, and so the gap between t_{pLH} and t_{pHL} tends to widen. As a student, you'll have the privilege of carrying out this sort of analysis in Drills 5.8–5.13.

The analysis gets more complicated if there are MSI devices in the delay path, or if there are multiple paths from a given input signal to a given output signal. Thus, in large circuits, analysis of all of the different delay paths and transition directions can be very complex.

To permit a simplified "worst-case" analysis, designers often use a single *worst-case delay* specification that is the maximum of t_{pLH} and t_{pHL} specifications. The worst-case delay through a circuit is then computed as the sum of the worst-case delays through the individual components, independent of the transition direction and other circuit conditions. This may give an overly pessimistic view of the overall circuit delay, but it saves design time and it's guaranteed to work.

worst-case delay

5.2.5 Timing Analysis Tools

Sophisticated CAD tools for logic design make timing analysis even easier. Their component libraries typically contain not only the logic symbols and functional models for various logic elements, but also their timing models. A simulator allows you to apply input sequences and observe how and when outputs are produced in response. You typically can control whether minimum, typical, maximum, or some combination of delay values are used.

Even with a simulator, you're not completely off the hook. It's usually up the designer to supply the input sequences for which the simulator should produce outputs. Thus, you'll need to have a good feel for what to look for and how to stimulate your circuit to produce and observe the worst-case delays.

Some timing analysis programs can automatically find all possible delay paths in a circuit, and print out a sorted list of them, starting with the slowest. These results may be overly pessimistic, however, as some paths may actually not be used in normal operations of the circuit; the designer must still use some intelligence to interpret the results properly.

5.3 Combinational PLDs

5.3.1 Programmable Logic Arrays

Historically, the first PLDs were *programmable logic arrays (PLAs)*. A PLA is a combinational, two-level AND-OR device that can be programmed to realize any sum-of-products logic expression, subject to the size limitations of the device. Limitations are

programmable logic array (PLA)

- the number of inputs (n),
- the number of outputs (m), and
- the number of product terms (p).

inputs

outputs

product terms

We might describe such a device as "an $n \times m$ PLA with p product terms." In general, p is far less than the number of n-variable minterms (2^n). Thus, a PLA cannot perform arbitrary n-input, m-output logic functions; its usefulness is limited to functions that can be expressed in sum-of-products form using p or fewer product terms.

An $n \times m$ PLA with p product terms contains p $2n$-input AND gates and m p-input OR gates. Figure 5-21 shows a small PLA with four inputs, six AND

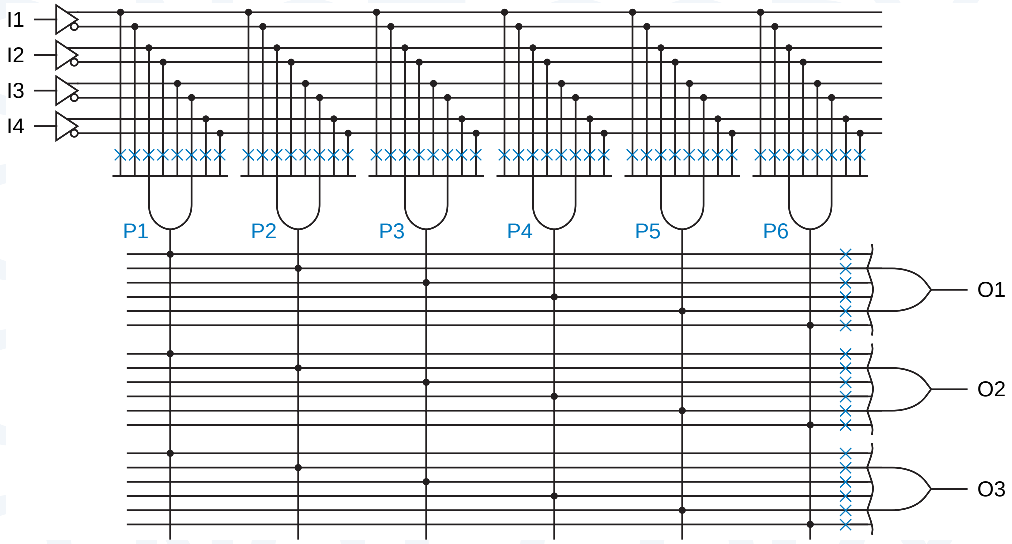


Figure 5-21 A 4 × 3 PLA with six product terms.

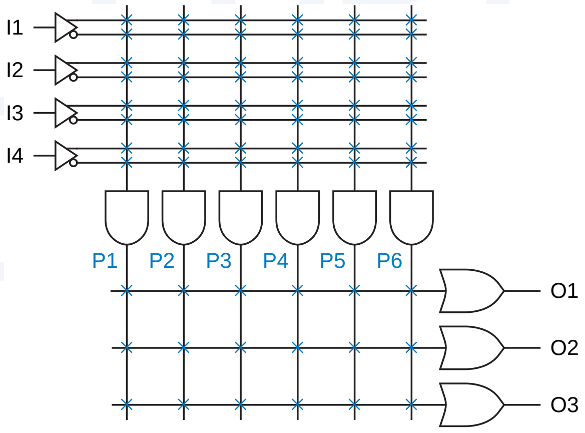
gates, and three OR gates and outputs. Each input is connected to a buffer that produces both a true and a complemented version of the signal for use within the array. Potential connections in the array are indicated by X's; the device is programmed by establishing only the connections that are actually needed. The needed connections are made by *fuses*, which are actual fusible links or non-volatile memory cells, depending on technology as we explain in Sections 5.3.4 and 5.3.5. Thus, each AND gate's inputs can be any subset of the primary input signals and their complements. Similarly, each OR gate's inputs can be any subset of the AND-gate outputs.

PLD fuses

PLA diagram

As shown in Figure 5-22, a more compact diagram can be used to represent a PLA. Moreover, the layout of this diagram more closely resembles the actual internal layout of a PLA chip (e.g., Figure 5-28 on page 308).

Figure 5-22 Compact representation of a 4 × 3 PLA with six product terms.



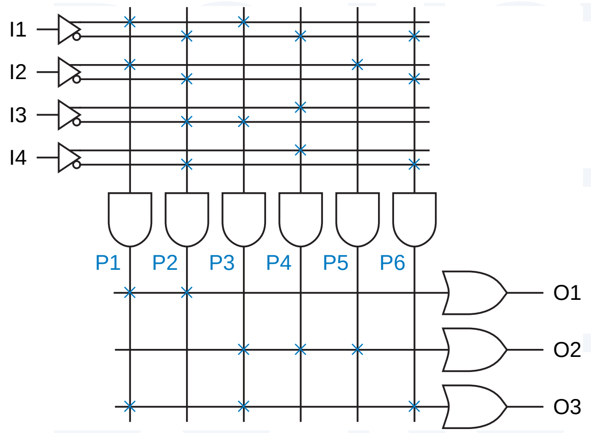


Figure 5-23
A 4 × 3 PLA programmed with a set of three logic equations.

The PLA in Figure 5-22 can perform any three 4-input combinational logic functions that can be written as sums of products using a total of six or fewer distinct product terms, for example:

$$\begin{aligned} O1 &= I1 \cdot I2 + I1' \cdot I2' \cdot I3' \cdot I4' \\ O2 &= I1 \cdot I3' + I1' \cdot I3 \cdot I4 + I2 \\ O3 &= I1 \cdot I2 + I1 \cdot I3' + I1' \cdot I2' \cdot I4' \end{aligned}$$

These equations have a total of eight product terms, but the first two terms in the O3 equation are the same as the first terms in the O1 and O2 equations. The programmed connection pattern in Figure 5-23 matches these logic equations.

PLA constant outputs

Sometimes a PLA output must be programmed to be a constant 1 or a constant 0. That's no problem, as shown in Figure 5-24. Product term P1 is always 1 because its product line is connected to no inputs and is therefore always pulled HIGH; this constant-1 term drives the O1 output. No product term drives the O2 output, which is therefore always 0. Another method of obtaining a constant-0 output is shown for O3. Product term P2 is connected to each input variable and its complement; therefore, it's always 0 ($X \cdot X' = 0$).

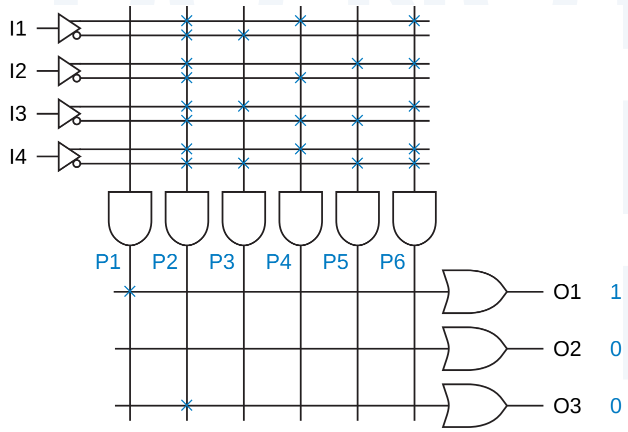


Figure 5-24
A 4 × 3 PLA programmed to produce constant 0 and 1 outputs.

AN UNLIKELY GLITCH

Theoretically, if *all* of the input variables in Figure 5-24 change *simultaneously*, the output of product term P2 could have a brief 0-1-0 glitch. This is highly unlikely in typical applications, and is impossible if one input happens to be unused and is connected to a constant logic signal.

Our example PLA has too few inputs, outputs, and AND gates (product terms) to be very useful. An n-input PLA could conceivably use as many as 2^n product terms, to realize all possible n-variable minterms. The actual number of product terms in typical commercial PLAs is far fewer, on the order of 4 to 16 per output, regardless of the value of n.

The Signetics 82S100 was a typical example of the PLAs that were introduced in the mid-1970s. It had 16 inputs, 48 AND gates, and 8 outputs. Thus, it had $2 \times 16 \times 48 = 1536$ fuses in the AND array and $8 \times 48 = 384$ in the OR array. Off-the-shelf PLAs like the 82S100 have since been supplanted by PALs, CPLDs, and FPGAs, but custom PLAs are often synthesized to perform complex combinational logic within a larger ASIC.

5.3.2 Programmable Array Logic Devices

programmable array logic (PAL) device

A special case of a PLA, and today's most commonly used type of PLD, is the *programmable array logic (PAL) device*. Unlike a PLA, in which both the AND and OR arrays are programmable, a PAL device has a *fixed* OR array.

The first PAL devices used TTL-compatible bipolar technology and were introduced in the late 1970s. Key innovations in the first PAL devices, besides the introduction of a catchy acronym, were the use of a fixed OR array and bidirectional input/output pins.

PAL16L8

These ideas are well illustrated by the *PAL16L8*, shown in Figures 5-25 and 5-26 and one of today's most commonly used combinational PLD structures. Its programmable AND array has 64 rows and 32 columns, identified for programming purposes by the small numbers in the figure, and $64 \times 32 = 2048$ fuses. Each of the 64 AND gates in the array has 32 inputs, accommodating 16 variables and their complements; hence, the "16" in "PAL16L8".

FRIENDS AND FOES

PAL is a registered trademark of Advanced Micro Devices, Inc. Like other trademarks, it should be used only as an adjective. Use it as a noun or without a trademark notice at your own peril (as I learned in a letter from AMD's lawyers in February 1989).

To get around AMD's trademark, I suggest that you use a descriptive name that is more indicative of the device's internal structure: a *fixed-OR element (FOE).*

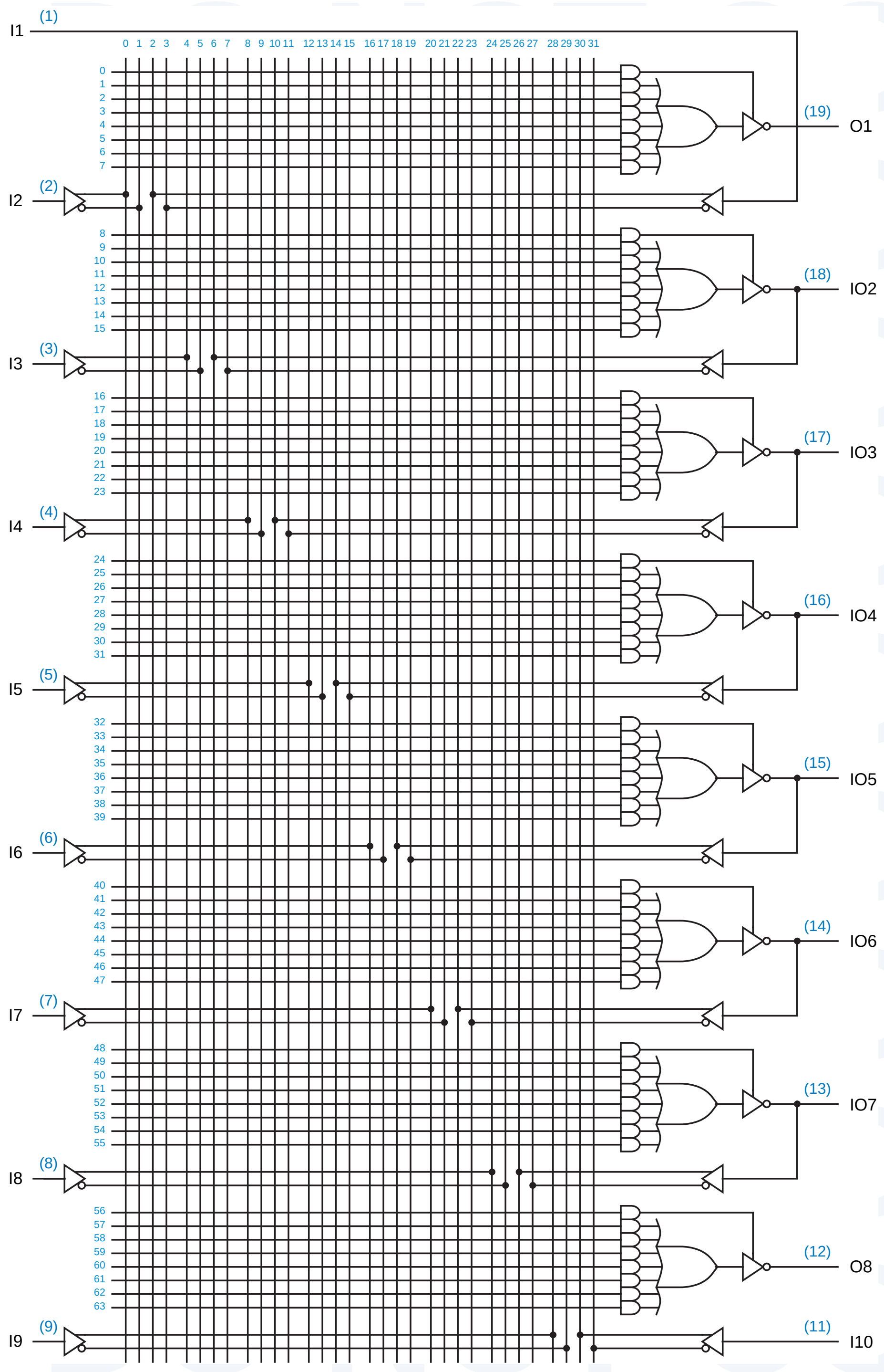


Figure 5-25 Logic diagram of the PAL16L8.

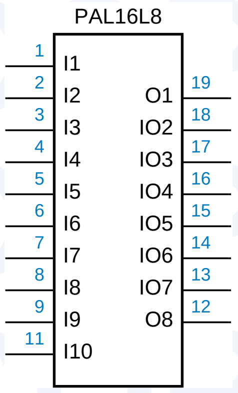


Figure 5-26
Traditional logic symbol for the PAL16L8.

Eight AND gates are associated with each output pin of the PAL16L8. Seven of them provide inputs to a fixed 7-input OR gate. The eighth, which we call the *output-enable gate*, is connected to the three-state enable input of the output buffer; the buffer is enabled only when the output-enable gate has a 1 output. Thus, an output of the PAL16L8 can perform only logic functions that can be written as sums of seven or fewer product terms. Each product term can be a function of any or all 16 inputs, but only seven such product terms are available.

output-enable gate

Although the PAL16L8 has up to 16 inputs and up to 8 outputs, it is housed in a dual in-line package with only 20 pins, including two for power and ground (the corner pins, 10 and 20). This magic is the result of six bidirectional pins (13–18) that may be used as inputs or outputs or both. This and other differences between the PAL16L8 and a PLA structure are summarized below:

- The PAL16L8 has a fixed OR array, with seven AND gates permanently connected to each OR gate. AND-gate outputs cannot be shared; if a product term is needed by two OR gates, it must be generated twice.
- Each output of the PAL16L8 has an individual three-state output enable signal, controlled by a dedicated AND gate (the output-enable gate). Thus, outputs may be programmed as always enabled, always disabled, or enabled by a product term involving the device inputs.

HOW USEFUL ARE SEVEN PRODUCT TERMS?

The worst-case logic function for two-level AND-OR design is an n-input XOR (parity) function, which requires 2^{n-1} product terms. However, less perverse functions with more than seven product terms of a PAL16L8 can often be built by decomposing them into a 4-level structure (AND-OR-AND-OR) that can be realized with two passes through the AND-OR array. Unfortunately, besides using up PLD outputs for the first-pass terms, this doubles the delay, since a first-pass input must pass through the PLD twice to propagate to the output.

COMBINATIONAL, NOT COMBINATORIAL!

A step *backwards* in MMI's introduction of PAL devices was their popularization of the word "combinatorial" to describe combinational circuits. *Combinational* circuits have no memory—their output at any time depends on the current input *combination*. For well-rounded computer engineers, the word "combinatorial" should conjure up vivid images of binomial coefficients, problem-solving complexity, and computer-science-great Donald Knuth.

- There is an inverter between the output of each OR gate and the external pin of the device.
- Six of the output pins, called *I/O pins*, may also be used as inputs. This provides many possibilities for using each I/O pin, depending on how the device is programmed:

  I/O pin

  – If an I/O pin's output-control gate produces a constant 0, then the output is always disabled and the pin is used strictly as an input.
  – If the input signal on an I/O pin is not used by any gates in the AND array, then the pin may be used strictly as an output. Depending on the programming of the output-enable gate, the output may always be enabled, or it may be enabled only for certain input conditions.
  – If an I/O pin's output-control gate produces a constant 1, then the output is always enabled, but the pin may still be used as an input too. In this way, outputs can be used to generate first-pass "helper terms" for logic functions that cannot be performed in a single pass with the limited number of AND terms available for a single output. We'll show an example of this case on page 325.
  – In another case with an I/O pin always output-enabled, the output may be used as an input to AND gates that affect the very same output. That is, we can embed a feedback sequential circuit in a PAL16L8. We'll discuss this case in \secref{palatch}.

PAL16L8

The *PAL20L8* is another combinational PLD similar to the PAL16L8, except that its package has four more input-only pins and each of its AND gates has eight more inputs to accommodate them. Its output structure is the same as the PAL16L8's.

5.3.3 Generic Array Logic Devices

In \chapref{SeqPLDs} we'll introduce sequential PLDs, programmable logic devices that provide flip-flops at some or all OR-gate outputs. These devices can be programmed to perform a variety of useful sequential-circuit functions.

COMBINATIONAL PLD SPEED

The speed of a combinational PLD is usually stated as a single number giving the propagation delay t_{PD} from any input to any output for either direction of transition. PLDs are available in a variety of speed grades; commonly used parts run at 10 ns. In 1998, the fastest available combinational PLDs included a bipolar PAL16L8 at 5 ns and a 3.3-V CMOS GAL22LV10 at 3.5 ns.

generic array logic
GAL device
GAL16V8

One type of sequential PLD, first introduced by Lattice Semiconductor, is called *generic array logic* or a *GAL device*, and is particularly popular. A single GAL device type, the *GAL16V8*, can be configured (via programming and a corresponding fuse pattern) to emulate the AND-OR, flip-flop, and output structure of any of a variety of combinational and sequential PAL devices, including the PAL16L8 introduced previously. What's more, the GAL device can be erased electrically and reprogrammed.

GAL16V8C

Figure 5-27 shows the logic diagram for a GAL16V8 when it has been configured as a strictly combinational device similar to the PAL16L8. This configuration is achieved by programming two "architecture-control" fuses, not shown. In this configuration, the device is called a *GAL16V8C*.

output polarity

The most important thing to note about the GAL16V8C logic diagram, compared to that of a PAL16L8 on page 303, is that an XOR gate has been inserted between each OR output and the three-state output driver. One input of the XOR gate is "pulled up" to a logic 1 value but connected to ground (0) via a fuse. If this fuse is intact, the XOR gate simply passes the OR-gate's output unchanged, but if the fuse is blown the XOR gate inverts the OR-gate's output. This fuse is said to control the *output polarity* of the corresponding output pin.

Output-polarity control is a very important feature of modern PLDs, including the GAL16V8. As we discussed in Section 4.6.2, given a logic function to minimize, an ABEL compiler finds minimal sum-of-products expressions for both the function and its complement. If the complement yields fewer product terms, it can be used if the GAL16V8's output polarity fuse is set to invert. Unless overridden, the compiler automatically makes the best selection and sets up the fuse patterns appropriately.

PALCE16V8
GAL20V8
PALCE20V8

Several companies make a part that is equivalent to the GAL16V8, called the *PALCE16V8*. There is also a 24-pin GAL device, the *GAL20V8* or *PALCE20V8*, that can be configured to emulate the structure of the PAL20L8 or any of a variety of sequential PLDs, as described in \secref{seqGAL}.

LEGAL NOTICE

GAL is a trademark of Lattice Semiconductor, Hillsboro, OR 97124.

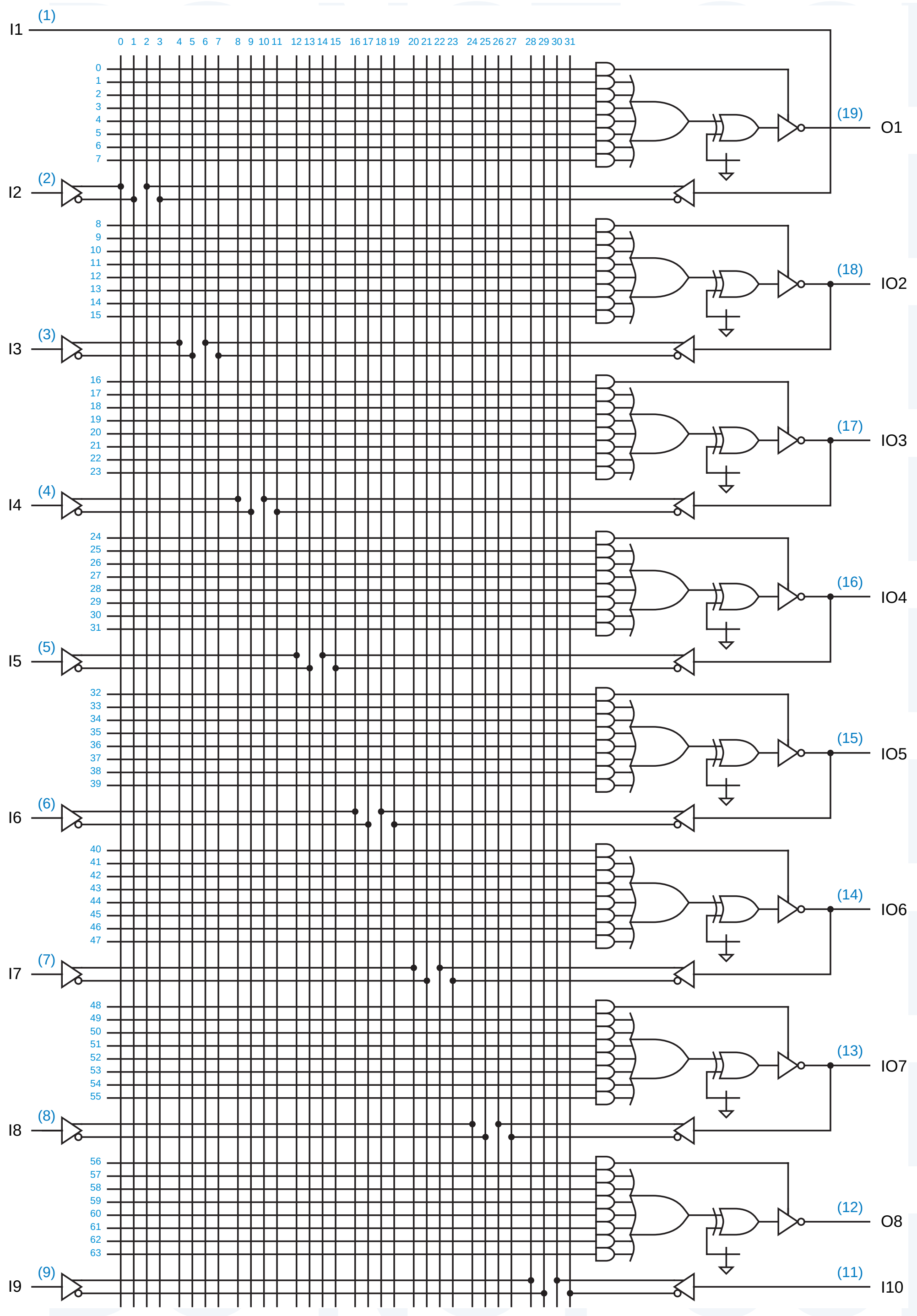


Figure 5-27 Logic diagram of the GAL16V8C.

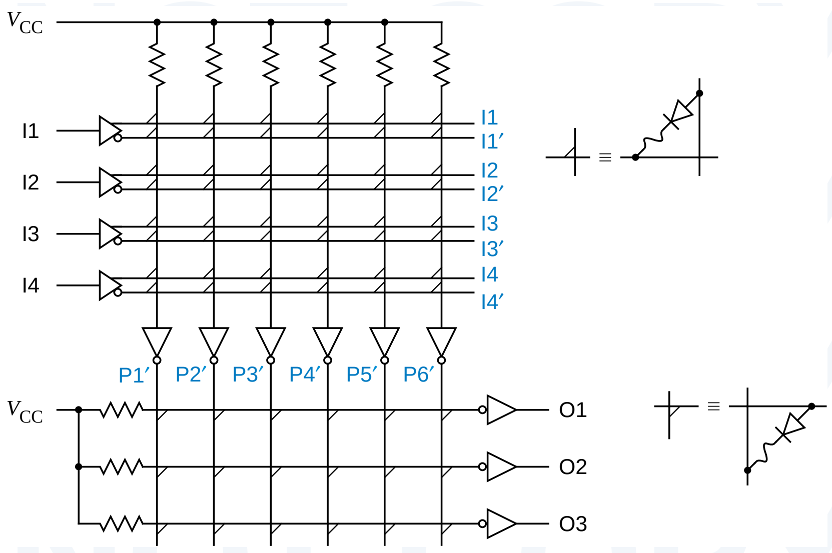


Figure 5-28
A 4 × 3 PLA built using TTL-like open-collector gates and diode logic.

*5.3.4 Bipolar PLD Circuits

There are several different circuit technologies for building and physically programming a PLD. Early commercial PLAs and PAL devices used bipolar circuits. For example, Figure 5-28 shows how the example 4 × 3 PLA circuit of the preceding section might be built in a bipolar, TTL-like technology. Each potential connection is made by a diode in series with a metal link that may be present or absent. If the link is present, then the diode connects its input into a diode-AND function. If the link is missing, then the corresponding input has no effect on that AND function.

AND plane

A diode-AND function is performed because each and every horizontal "input" line that is connected via a diode to a particular vertical "AND" line must be HIGH in order for that AND line to be HIGH. If an input line is LOW, it pulls LOW all of the AND lines to which it is connected. This first matrix of circuit elements that perform the AND function is called the *AND plane.*

OR plane

Each AND line is followed by an inverting buffer, so overall a NAND function is obtained. The outputs of the first-level NAND functions are combined by another set of programmable diode AND functions, once again followed by inverters. The result is a two-level NAND-NAND structure that is functionally equivalent to the AND-OR PLA structure described in the preceding section. The matrix of circuit elements that perform the OR function (or the second NAND function, depending on how you look at it) is called the *OR plane*.

fusible link

A bipolar PLD chip is manufactured with all of its diodes present, but with a tiny *fusible link* in series with each one (the little squiggles in Figure 5-28). By

* Throughout this book, optional sections are marked with an asterisk.

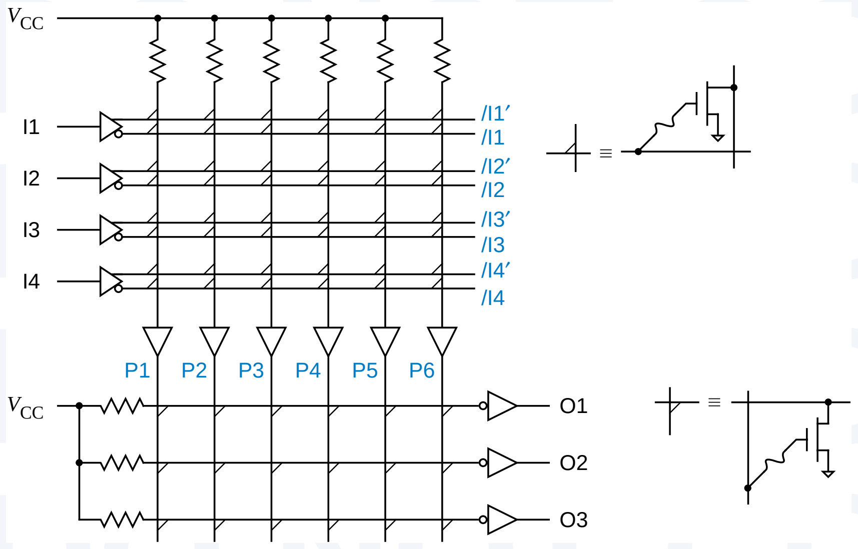


Figure 5-29
A 4 × 3 PLA built using CMOS logic.

applying special input patterns to the device, it is possible to select individual links, apply a high voltage (10–30 V), and thereby vaporize selected links.

Early bipolar PLDs had reliability problems. Sometimes the stored patterns changed because of incompletely vaporized links that would “grow back,” and sometimes intermittent failures occurred because of floating shrapnel inside the IC package. However, these problems have been largely worked out, and reliable fusible-link technology is used in today’s bipolar PLDs.

*5.3.5 CMOS PLD Circuits

Although they’re still available, bipolar PLDs have been largely supplanted by CMOS PLDs with a number of advantages, including reduced power consumption and reprogrammability. Figure 5-29 shows a CMOS design for the 4 × 3 PLA circuit of Section 5.3.1.

Instead of a diode, an *n*-channel transistor with a programmable connection is placed at each intersection between an input line and a word line. If the input is LOW, then the transistor is “off,” but if the input is HIGH, then the transistor is “on,” which pulls the AND line LOW. Overall, an inverted-input AND (i.e., NOR) function is obtained. This is similar in structure and function to a normal CMOS *k*-input NOR gate, except that the usual series connection of *k* *p*-channel pull-up transistors has been replaced with a passive pull-up resistor (in practice, the pull-up is a single *p*-channel transistor with a constant bias).

As shown in color on Figure 5-29, the effects of using an *inverted*-input AND gate are canceled by using the opposite (complemented) input lines for each input, compared with Figure 5-28. Also notice that the connection between

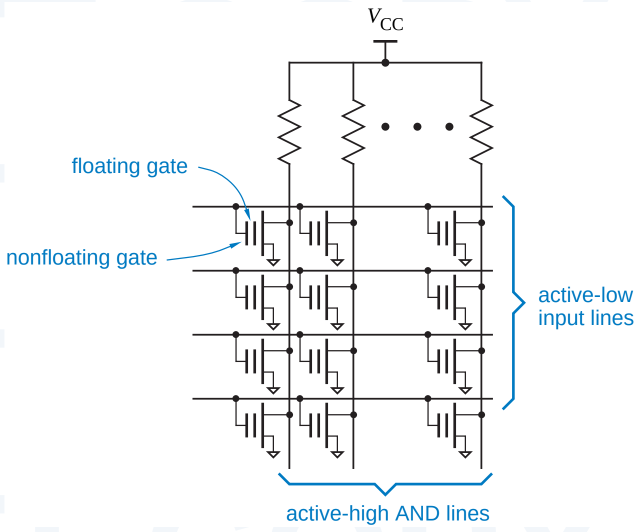


Figure 5-30 AND plane of an EPLD using floating-gate MOS transistors.

the AND plane and the OR plane is noninverting, so the AND plane performs a true AND function.

The outputs of the first-level AND functions are combined in the OR plane by another set of NOR functions with programmable connections. The output of each NOR function is followed by an inverter, so a true OR function is realized, and overall the PLA performs an AND-OR function as desired.

In CMOS PLD technologies, the programmable links shown in Figure 5-29 are not normally fuses. In non-field-programmable devices, such as custom VLSI chips, the presence or absence of each link is simply established as part of the metal mask pattern for the manufacture of the device. By far the most common programming technology, however, is used in CMOS EPLDs, as discussed next.

erasable programmable logic device (EPLD)

An *erasable programmable logic device (EPLD)* can be programmed with any desired link configuration, as well as "erased" to its original state, either electronically or by exposing it to ultraviolet light. No, erasing does not cause links to suddenly appear or disappear! Rather, EPLDs use a different technology, called "floating-gate MOS."

floating-gate MOS transistor

As shown in Figure 5-30, an EPLD uses *floating-gate MOS transistors*. Such a transistor has two gates. The "floating" gate is unconnected and is surrounded by extremely high-impedance insulating material. In the original, manufactured state, the floating gate has no charge on it and has no effect on circuit operation. In this state, all transistors are effectively "connected"; that is, there is a logical link present at every crosspoint in the AND and OR planes.

To program an EPLD, the programmer applies a high voltage to the non-floating gate at each location where a logical link is not wanted. This causes a temporary breakdown in the insulating material and allows a negative charge to accumulate on the floating gate. When the high voltage is removed, the negative charge remains on the floating gate. During subsequent operations, the negative charge prevents the transistor from turning "on" when a HIGH signal is applied to the nonfloating gate; the transistor is effectively disconnected from the circuit.

EPLD manufacturers claim that a properly programmed bit will retain 70% of its charge for at least 10 years, even if the part is stored at 125°C, so for most applications the programming can be considered to be permanent. However, EPLDs can also be erased.

electrically erasable PLD

Although some early EPLDs were packaged with a transparent lid and used light for erasing, today's devices are popular are *electrically erasable PLDs*. The floating gates in an electrically erasable PLD are surrounded by an extremely thin insulating layer, and can be erased by applying a voltage of the opposite polarity as the charging voltage to the nonfloating gate. Thus, the same piece of equipment that is normally used to program a PLD can also be used to erase an EPLD before programming it.

field-programmable gate array (FPGA)

Larger-scale, "complex" PLDs (CPLDs), also use floating-gate programming technology. Even larger devices, often called *field-programmable gate arrays (FPGAs)*, use read/write memory cells to control the state of each connection. The read/write memory cells are volatile—they do not retain their state when power is removed. Therefore, when power is first applied to the FPGA, all of its read/write memory must be initialized to a state specified by a separate, external nonvolatile memory. This memory is typically either a programmable read-only memory (PROM) chip attached directly to the FPGA or it's part of a microprocessor subsystem that initializes the FPGA as part of overall system initialization.

*5.3.6 Device Programming and Testing

PLD programmer
PROM programmer

A special piece of equipment is used to vaporize fuses, charge up floating-gate transistors, or do whatever else is required to program a PLD. This piece of equipment, found nowadays in almost all digital design labs and production facilities, is called a *PLD programmer* or a *PROM programmer*. (It can be used

CHANGING HARDWARE ON THE FLY

PROMs are normally used to supply the connection pattern for a read/write-memory-based FPGA, but there are also applications where the pattern is actually read from a floppy disk. You just received a floppy with a new software version? Guess what, you just got a new hardware version too!

This concept leads us to the intriguing idea, already being applied in some applications, of "reconfigurable hardware," where a hardware subsystem is redefined, on the fly, to optimize its performance for the particular task at hand.

with programmable read-only memories, “PROMs,” as well as for PLDs.) A typical PLD programmer includes a socket or sockets that physically accept the devices to be programmed, and a way to “download” desired programming patterns into the programmer, typically by connecting the programmer to a PC.

PLD programmers typically place a PLD into a special mode of operation in order to program it. For example, a PLD programmer typically programs the PLDs described in this chapter eight fuses at a time as follows:

1. Raise a certain pin to a predetermined, high voltage (such as 14 V) to put the device into programming mode.
2. Select a group of eight fuses by applying a binary “address” to certain inputs of the device. (For example, the 82S100 has 1920 fuses, and would therefore require 8 inputs to select one of 240 groups of 8 fuses.)
3. Apply an 8-bit value to the *outputs* of the device to specify the desired programming for each fuse (the outputs are used as inputs in programming mode).
4. Raise a second predetermined pin to the high voltage for a predetermined length of time (such as 100 microseconds) to program the eight fuses.
5. Lower the second predetermined pin to a low voltage (such as 0 V) to read out and verify the programming of the eight fuses.
6. Repeat steps 1–5 for each group of eight fuses.

in-system programmability

JTAG port

Many PLDs, especially larger CPLDs, feature *in-system programmability*. This means that the device can be programmed after it is already soldered into the system. In this case, the fuse patterns are applied to the device serially using four extra signals and pins, called the *JTAG port*, defined by IEEE standard 1149.1. These signals are defined so that multiple devices on the same printed-circuit board can be “daisy chained” and selected and programmed during the board manufacturing process using just one JTAG port on a special connector. No special high-voltage power supply is needed; each device uses a charge-pump circuit internally to generate the high voltage needed for programming.

As noted in step 5 above, fuse patterns are verified as they are programmed into a device. If a fuse fails to program properly the first time, the operation can be retried; if it fails to program properly after a few tries, the device is discarded (often with great prejudice and malice aforethought).

While verifying the fuse pattern of a programmed device proves that its fuses are programmed properly, it does not prove that the device will perform the logic function specified by those fuses. This is true because the device may have unrelated internal defects such as missing connections between the fuses and elements of the AND-OR array.

The only way to test for all defects is to put the device into its normal operational mode, apply a set of normal logic inputs, and observe the outputs. The required input and output patterns, called test vectors, can be specified by the designer as we showed in Section 4.6.7,or can be generated automatically by a special test-vector-generation program. Regardless of how the test vectors are generated, most PLD programmers have the ability to apply test-vector inputs to a PLD and to check its outputs against the expected results.

Most PLDs have a *security fuse* which, when programmed, disables the ability to read fuse patterns from the device. Manufacturers can program this fuse to prevent others from reading out the PLD fuse patterns in order to copy the product design. Even if the security fuse is programmed, test vectors still work, so the PLD can still be checked.

security fuse

5.4 Decoders

A *decoder* is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. The input code generally has fewer bits than the output code, and there is a one-to-one mapping from input code words into output code words. In a *one-to-one mapping*, each input code word produces a different output code word.

decoder

one-to-one mapping

The general structure of a decoder circuit is shown in Figure 5-31. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single, “disabled,” output code word.

The most commonly used input code is an n-bit binary code, where an n-bit word represents one of 2^n different coded values, normally the integers from 0 through 2^n–1. Sometimes an n-bit binary code is truncated to represent fewer than 2^n values. For example, in the BCD code, the 4-bit combinations 0000 through 1001 represent the decimal digits 0–9, and combinations 1010 through 1111 are not used.

The most commonly used output code is a 1-out-of-m code, which contains m bits, where one bit is asserted at any time. Thus, in a 1-out-of-4 code with active-high outputs, the code words are 0001, 0010, 0100, and 1000. With active-low outputs, the code words are 1110, 1101, 1011, and 0111.

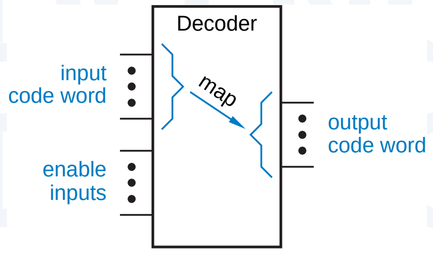


Figure 5-31
Decoder circuit structure.

Table 5-4
Truth table for a 2-to-4 binary decoder.

Inputs			Outputs			
EN	I1	I0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

5.4.1 Binary Decoders

binary decoder

The most common decoder circuit is an n-to-2^n decoder or *binary decoder*. Such a decoder has an n-bit binary input code and a 1-out-of-2^n output code. A binary decoder is used when you need to activate exactly one of 2^n outputs based on an n-bit input value.

For example, Figure 5-32(a) shows the inputs and outputs and Table 5-4 is the truth table of a 2-to-4 decoder. The input code word 1,I0 represents an integer in the range 0–3. The output code word Y3,Y2,Y1,Y0 has Y*i* equal to 1 if and only if the input code word is the binary representation of *i* and the *enable input* EN is 1. If EN is 0, then all of the outputs are 0. A gate-level circuit for the 2-to-4 decoder is shown in Figure 5-32(b). Each AND gate *decodes* one combination of the input code word I1,I0.

enable input

decode

The binary decoder's truth table introduces a "don't-care" notation for input combinations. If one or more input values do not affect the output values for some combination of the remaining inputs, they are marked with an "x" for that input combination. This convention can greatly reduce the number of rows in the truth table, as well as make the functions of the inputs more clear.

The input code of an n-bit binary decoder need not represent the integers from 0 through 2^n–1. For example, Table 5-5 shows the 3-bit Gray-code output

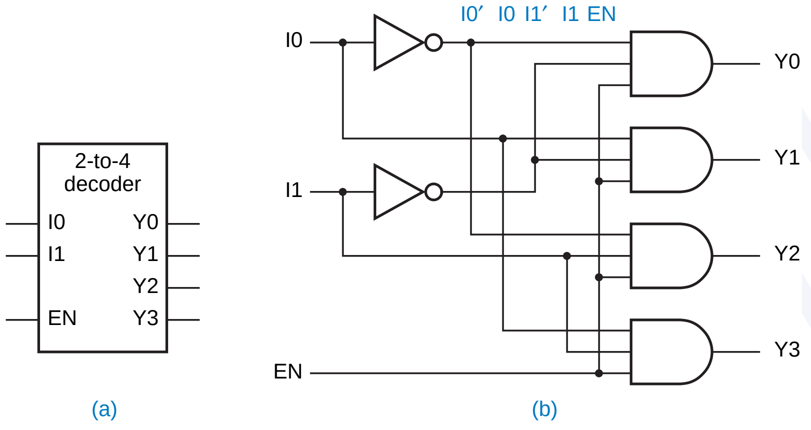


Figure 5-32
A 2-to-4 decoder:
(a) inputs and outputs;
(b) logic diagram.

Disk Position	I2	I1	I0	*Binary Decoder Output*
0°	0	0	0	Y0
45°	0	0	1	Y1
90°	0	1	1	Y3
135°	0	1	0	Y2
180°	1	1	0	Y6
225°	1	1	1	Y7
270°	1	0	1	Y5
315°	1	0	0	Y4

Table 5-5 Position encoding for a 3-bit mechanical encoding disk.

of a mechanical encoding disk with eight positions. The eight disk positions can be decoded with a 3-bit binary decoder with the appropriate assignment of signals to the decoder outputs, as shown in Figure 5-33.

Also, it is not necessary to use all of the outputs of a decoder, or even to decode all possible input combinations. For example, a *decimal* or *BCD decoder* decodes only the first ten binary input combinations 0000–1001 to produce outputs Y0–Y9.

decimal decoder
BCD decoder

5.4.2 Logic Symbols for Larger-Scale Elements

Before describing some commercially available 74-series MSI decoders, we need to discuss general guidelines for drawing logic symbols for larger-scale logic elements.

The most basic rule is that logic symbols are drawn with inputs on the left and outputs on the right. The top and bottom edges of a logic symbol are not normally used for signal connections. However, explicit power and ground connections are sometimes shown at the top and bottom, especially if these connections are made on “nonstandard” pins. (Most MSI parts have power and ground connected to the corner pins, e.g., pins 8 and 16 of a 16-pin DIP package.)

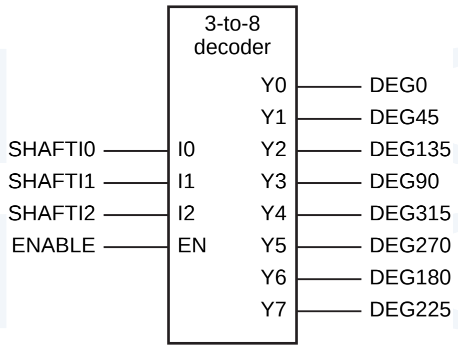


Figure 5-33 Using a 3-to-8 binary decoder to decode a Gray code.

LOGIC FAMILIES

Most logic gates and larger-scale elements are available in a variety of CMOS and families, many of which we described in Sections 3.8 and 3.11. For example, the 74LS139, 74S139, 74ALS139, 74AS139, 74F139, 74HC139, 74HCT139, 74ACT139, 74AC139, 74FCT139 74AHC139, 74AHCT139, 74LC139, 74LVC139, and 74VHC139 are all dual 2-to-4 decoders with the same logic function, but in electrically different TTL and CMOS families and sometimes in different packages. In addition, "macro" logic elements with the same pin names and functions as the '139 and other popular 74-series devices are available as building blocks in most FPGA and ASIC design environments.

Throughout this text, we use "74x" as a generic prefix. And we'll sometimes omit the prefix and write, for example, '139. In a real schematic diagram for a circuit that you are going to build or simulate, you should include the full part number, since timing, loading, and packaging characteristics depend on the family.

Like gate symbols, the logic symbols for larger-scale elements associate an active level with each pin. With respect to active levels, it's important to use a consistent convention to naming the internal signals and external pins.

Larger-scale elements almost always have their signal names defined in terms of the functions performed *inside* their symbolic outline, as explained in Section 5.1.4. For example, Figure 5-34(a) shows the logic symbol for one section of a 74x139 dual 2-to-4 decoder, an MSI part that we'll fully describe in the next subsection. When the G input is asserted, one of the outputs Y0–Y3 is asserted, as selected by a 2-bit code applied to the A and B inputs. It is apparent from the symbol that the G input pin and all of the output pins are active low.

When the 74x139 symbol appears in the logic diagram for a real application, its inputs and outputs have signals connected to other devices, and each such signal has a name that indicates its function in the application. However, when we describe the 74x139 in isolation, we might still like to have a name for the signal on each external pin. Figure 5-34(b) shows our naming convention in this case. Active-high pins are given the same name as the internal signal, while active-low pins have the internal signal name followed by the suffix "_L".

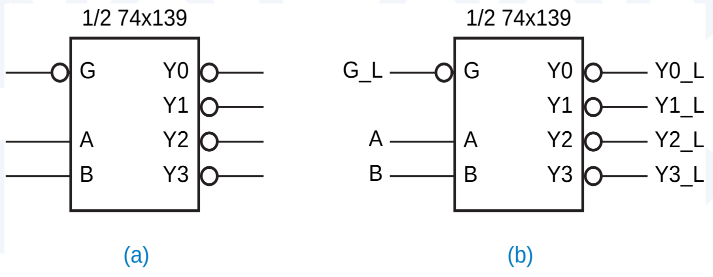


Figure 5-34 Logic symbol for one-half of a 74x139 dual 2-to-4 decoder: (a) conventional symbol; (b) default signal names associated with external pins.

IEEE STANDARD LOGIC SYMBOLS

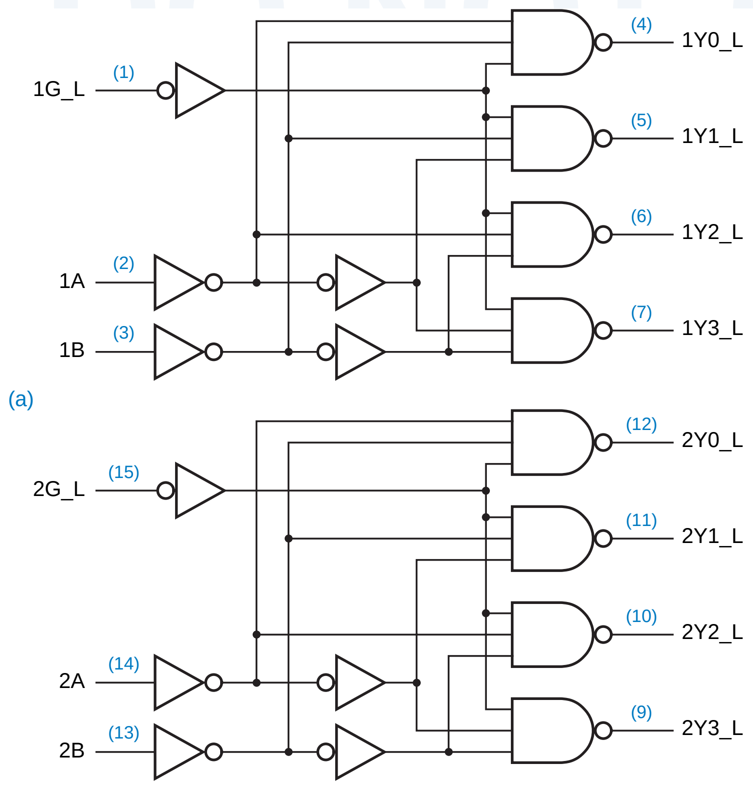
Throughout this book, we use “traditional” symbols for larger-scale logic elements. The IEEE standard uses somewhat different symbols for larger-scale logic elements. IEEE standard symbols, as well as the pros and cons of IEEE versus traditional symbols, are discussed in Appendix A.

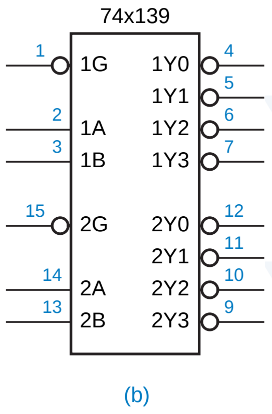
5.4.3 The 74x139 Dual 2-to-4 Decoder

74x139

Two independent and identical 2-to-4 decoders are contained in a single MSI part, the *74x139*. The gate-level circuit diagram for this IC is shown in Figure 5-35(a). Notice that the outputs and the enable input of the ’139 are active-low. Most MSI decoders were originally designed with active-low outputs, since TTL inverting gates are generally faster than noninverting ones. Also notice that the ’139 has extra inverters on its select inputs. Without these inverters, each select input would present three AC or DC loads instead of one, consuming much more of the fanout budget of the device that drives it.

Figure 5-35 The 74x139 dual 2-to-4 decoder: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol; (c) logic symbol for one decoder.





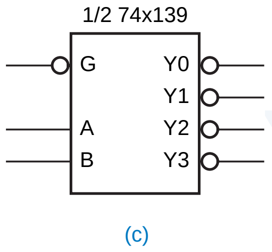


Table 5-6 Truth table for one-half of a 74x139 dual 2-to-4 decoder.

Inputs			*Outputs*			
G_L	B	A	Y3_L	Y2_L	Y1_L	Y0_L
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

A logic symbol for the 74x139 is shown in Figure 5-35(b). Notice that all of the signal names inside the symbol outline are active-high (no "_L"), and that inversion bubbles indicate active-low inputs and outputs. Often a schematic may use a generic symbol for just one decoder, one-half of a '139, as shown in (c). In this case, the assignment of the generic function to one half or the other of a particular '139 package can be deferred until the schematic is completed.

Table 5-6 is the truth table for a 74x139-type decoder. The truth tables in some manufacturers' data books use L and H to denote the input and output signal voltage levels, so there can be no ambiguity about the electrical function of the device; a truth table written this way is sometimes called a *function table*. However, since we use positive logic throughout this book, we can use 0 and 1 without ambiguity. In any case, the truth table gives the logic function in terms of the *external pins* of the device. A truth table for the function performed *inside* the symbol outline would look just like Table 5-4, except that the input signal names would be G, B, A.

function table

Some logic designers draw the symbol for 74x139s and other logic functions without inversion bubbles. Instead, they use an overbar on signal names inside the symbol outline to indicate negation, as shown in Figure 5-36(a). This notation is self-consistent, but it is inconsistent with our drawing standards for bubble-to-bubble logic design. The symbol shown in (b) is absolutely *incorrect*: according to this symbol, a logic 1, not 0, must be applied to the enable pin to enable the decoder.

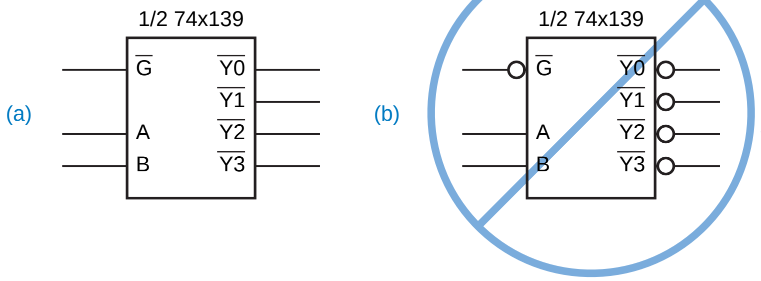


Figure 5-36 More ways to symbolize a 74x139: (a) correct but to be avoided; (b) incorrect because of double negations.

BAD NAMES Some manufacturers' data sheets have inconsistencies similar to Figure 5-36(b). For example, Texas Instruments' data sheet for the 74AHC139 uses active low-names like $\overline{\text{1G}}$ for the enable inputs, with the overbar indicating an active-low pin, but active-high names like 1Y0 for all the active-low output pins. On the other hand, Motorola's data sheet for the 74VHC139 correctly uses overbars on the names for both the enable inputs and the outputs, but the overbars are barely visible in the device's function table due to a typographical problem.

I've also had the personal experience of building a printed-circuit board with many copies of a new device from a vendor whose documentation clearly indicated that a particular input was active low, only to find out upon the first power-on that the input was active high.

The moral of the story is that you have to study the description of each device to know what's really going on. And if it's a brand-new device, whether from a commercial vendor or your own company's ASIC group, you should double-check all of the signal polarities and pin assignments before committing to a PCB. Be assured, however, that the signal names in this text *are* consistent and correct.

5.4.4 The 74x138 3-to-8 Decoder

74x138

The *74x138* is a commercially available MSI 3-to-8 decoder whose gate-level circuit diagram and symbol are shown in Figure 5-37; its truth table is given in Table 5-7. Like the 74x139, the 74x138 has active-low outputs, and it has three enable inputs (G1, /G2A, /G2B), all of which must be asserted for the selected output to be asserted.

The logic function of the '138 is straightforward—an output is asserted if and only if the decoder is enabled and the output is selected. Thus, we can easily write logic equations for an internal output signal such as Y5 in terms of the internal input signals:

$$\text{Y5} = \underbrace{\text{G1} \cdot \text{G2A} \cdot \text{G2B}}_{\text{enable}} \cdot \underbrace{\text{C} \cdot \text{B}' \cdot \text{A}}_{\text{select}}$$

However, because of the inversion bubbles, we have the following relations between internal and external signals:

$$\begin{aligned} \text{G2A} &= \text{G2A\_L}' \\ \text{G2B} &= \text{G2B\_L}' \\ \text{Y5} &= \text{Y5\_L}' \end{aligned}$$

Therefore, if we're interested, we can write the following equation for the external output signal Y5_L in terms of external input signals:

$$\begin{aligned} \text{Y5\_L} = \text{Y5}' &= (\text{G1} \cdot \text{G2A\_L}' \cdot \text{G2B\_L}' \cdot \text{C} \cdot \text{B}' \cdot \text{A})' \\ &= \text{G1}' + \text{G2A\_L} + \text{G2B\_L} + \text{C}' + \text{B} + \text{A}' \end{aligned}$$

Table 5-7 Truth table for a 74x138 3-to-8 decoder.

Inputs						*Outputs*							
G1	G2A_L	G2B_L	C	B	A	Y7_L	Y6_L	Y5_L	Y4_L	Y3_L	Y2_L	Y1_L	Y0_L
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

On the surface, this equation doesn't resemble what you might expect for a decoder, since it is a logical sum rather than a product. However, if you practice bubble-to-bubble logic design, you don't have to worry about this; you just give the output signal an active-low name and remember that it's active low when you connect it to other inputs.

5.4.5 Cascading Binary Decoders

Multiple binary decoders can be used to decode larger code words. Figure 5-38 shows how two 3-to-8 decoders can be combined to make a 4-to-16 decoder. The availability of both active-high and active-low enable inputs on the 74x138 makes it possible to enable one or the other directly based on the state of the most significant input bit. The top decoder (U1) is enabled when N3 is 0, and the bottom one (U2) is enabled when N3 is 1.

To handle even larger code words, binary decoders can be cascaded hierarchically. Figure 5-39 shows how to use half of a 74x139 to decode the two high-order bits of a 5-bit code word, thereby enabling one of four 74x138s that decode the three low-order bits.

5.4.6 Decoders in ABEL and PLDs

Nothing in logic design is much easier than writing the PLD equations for a decoder. Since the logic expression for each output is typically just a single product term, decoders are very easily targeted to PLDs and use few product-term resources.

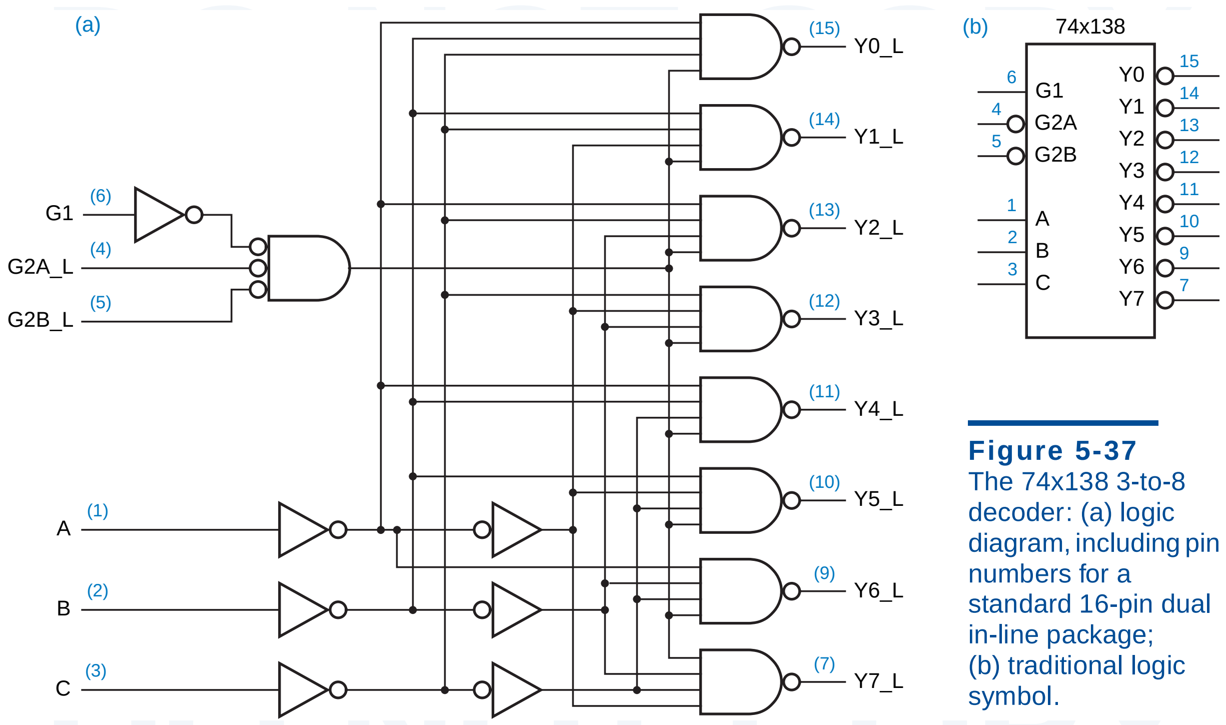


Figure 5-37
The 74x138 3-to-8 decoder: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

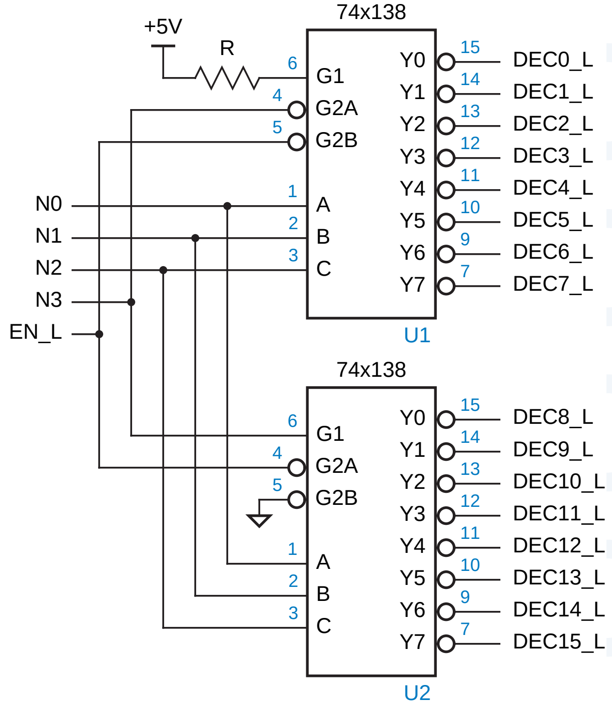


Figure 5-38
Design of a 4-to-16 decoder using 74x138s.

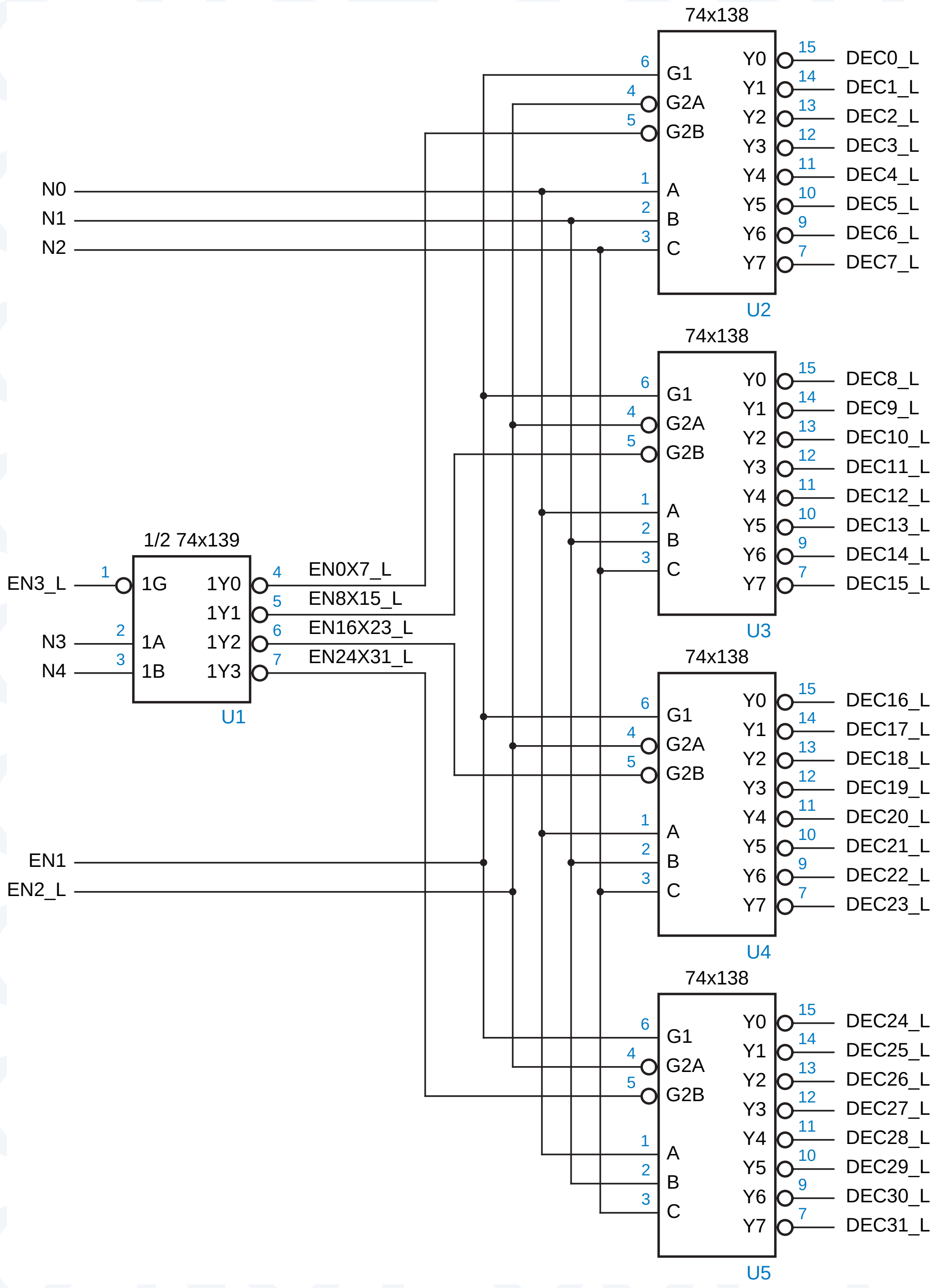


Figure 5-39 Design of a 5-to-32 decoder using 74x138s and a 74x139.

Table 5-8 An ABEL program for a 74x138-like 3-to-8 binary decoder.

```
module Z74X138
title '74x138 Decoder PLD
J. Wakerly, Stanford University'
Z74X138 device 'P16L8';

" Input pins
A, B, C, !G2A, !G2B, G1      pin 1, 2, 3, 4, 5, 6;

" Output pins
!Y0, !Y1, !Y2, !Y3           pin 19, 18, 17, 16 istype 'com';
!Y4, !Y5, !Y6, !Y7           pin 15, 14, 13, 12 istype 'com';

" Constant expression
ENB = G1 & G2A & G2B;

equations
Y0 = ENB & !C & !B & !A;
Y1 = ENB & !C & !B &  A;
Y2 = ENB & !C &  B & !A;
Y3 = ENB & !C &  B &  A;
Y4 = ENB &  C & !B & !A;
Y5 = ENB &  C & !B &  A;
Y6 = ENB &  C &  B & !A;
Y7 = ENB &  C &  B &  A;

end Z74X138
```

For example, Table 5-8 is an ABEL program for a 74x138-like 3-to-8 binary decoder as realized in a PAL16L8. A corresponding logic diagram with signal names is shown in Figure 5-40. In the ABEL program, notice the use of the “!” prefix in the pin declarations to specify active-low inputs and outputs, even though the equations are written in terms of active-high signals.

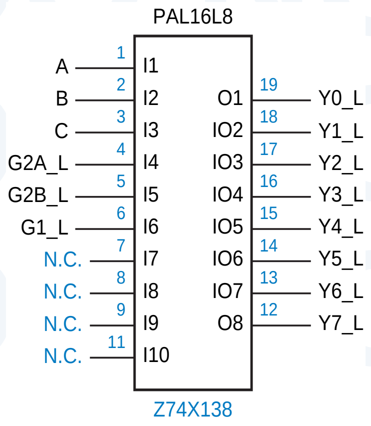


Figure 5-40 Logic diagram for the PAL16L8 used as a 74x138 decoder.

Also note that this example defines a constant expression for ENB. Here, ENB is not an input or output signal, but merely a user-defined name. In the equations section, the compiler substitutes the expression (G1 & G2A & G2B) everywhere that “ENB” appears. Assigning the constant expression to a user-defined name improves the program’s readability and maintainability.

If all you needed was a ’138, you’d be better off using a real ’138 than a PLD. However, if you need nonstandard functionality, then the PLD can usually achieve it much more cheaply and easily than an MSISSI-based solution. For example, if you need the functionality of a ’138 but with active-high outputs, you need only to change two lines in the pin declarations of Table 5-8:

```
Y0, Y1, Y2, Y3          pin 19, 18, 17, 16 istype ’com’;
Y4, Y5, Y6, Y7          pin 15, 14, 13, 12 istype ’com’;
```

Since each of the equations required a single product of six variables (including the three in the ENB expression), each complemented equation requires a *sum* of six product terms, less than the seven available in a PAL16L8. If you use a PAL16V8 or other device with output polarity selection, then the compiler selects non-inverted output polarity to use only one product term per output.

Another easy change is to provide alternate enable inputs that are ORed with the main enable inputs. To do this, you need only define additional pins and modify the definition of ENB:

```
EN1, !EN2               pin 7, 8;
...
ENB = G1 & G2A & G2B # EN1 # EN2;
```

This change expands the number of product terms per output to three, each having a form similar to

```
Y0 = G1 & G2A & G2B & !C & !B &!A
   # EN1 & !C & !B & !A
   # EN2 & !C & !B & !A;
```

(Remember that the PAL16L8 has a fixed inverter and the PAL16V8 has a selectable inverter between the AND-OR array and the output of the PLD, so the actual output is active low as desired.)

If you add the extra enables to the version of the program with active-high outputs, then the PLD must realize the complement of the sum-of-products expression above. It’s not immediately obvious how many product terms this expression will have, and whether it will fit in a PAL16L8, but we can use the ABEL compiler to get the answer for us:

```
!Y0 = C # B # A # !G2B & !EN1 & !EN2
                # !G2A & !EN1 & !EN2
                # !G1  & !EN1 & !EN2;
```

The expression has a total of six product terms, so it fits in a PAL16L8.

Table 5-9 ABEL program fragment showing two-pass logic.

```
...
" Output pins
!Y0, !Y1, !Y2, !Y3          pin 19, 18, 17, 16 istype 'com';
!Y4, !Y5, !Y6, ENB          pin 15, 14, 13, 12 istype 'com';

equations
ENB = G1 & G2A & G2B # EN1 # EN2;
Y0 = POL $ (ENB & !C & !B & !A);
...
```

As a final tweak, we can add an input to dynamically control whether the output is active high or active low, and modify all of the equations as follows:

```
POL                 pin 9;
...
Y0 = POL $ (ENB & !C & !B & !A);
Y1 = POL $ (ENB & !C & !B &  A);
...
Y7 = POL $ (ENB &  C &  B &  A);
```

As a result of the XOR operation, the number of product terms needed per output increases to 9, in either output-pin polarity. Thus, even a PAL16V8 cannot implement the function as written.

helper output

The function can still be realized if we create a *helper output* to reduce the product term explosion. As shown in Table 5-9, we allocate an output pin for the ENB expression, and move the ENB equation into the equations section of the program. This reduces the product-term requirement to 5 in either polarity.

helper output

Besides sacrificing a pin for the helper output, this realization has the disadvantage of being slower. Any changes in the inputs to the helper expression must propagate through the PLD twice before reaching the final output. This is called *two-pass logic*. Many PLD and FPGA synthesis tools can automatically

two-pass logic

Table 5-10 Truth table for a customized decoder function.

CS_L	RD_L	A2	A1	A0	***Output(s) to Assert***
1	x	x	x	x	none
x	1	x	x	x	none
0	0	0	0	0	BILL_L, MARY_L
0	0	0	0	1	MARY_L, KATE_L
0	0	0	1	0	JOAN_L
0	0	0	1	1	PAUL_L
0	0	1	0	0	ANNA_L
0	0	1	0	1	FRED_L
0	0	1	1	0	DAVE_L
0	0	1	1	1	KATE_L

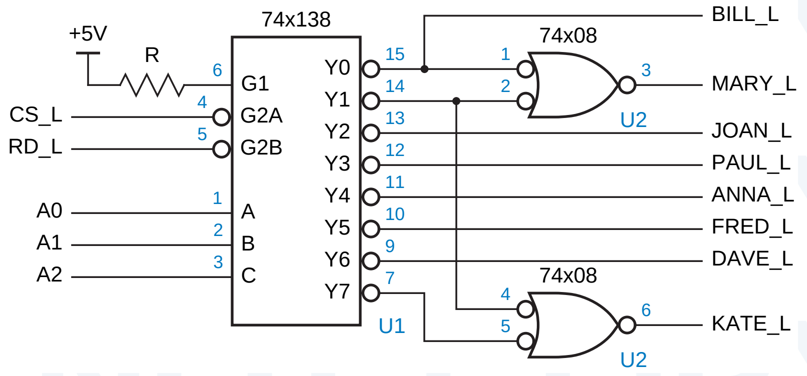


Figure 5-41 Customized decoder circuit.

generate logic with two or more passes if a required expression cannot be realized in just one pass through the logic array.

Decoders can be customized in other ways. A common customization is for a single output to decode more than one input combination. For example, suppose you needed to generate a set of enable signals according to Table 5-10 on the preceding page. A 74x138 MSI decoder can be augmented as shown in Figure 5-41 to perform the required function. This approach, while potentially less expensive than a PLD, has the disadvantages that it requires extra components and delay to create the required outputs, and it is not easily modified.

Table 5-11 ABEL equations for a customized decoder.

```
module CUSTMDEC
title 'Customized Decoder PLD
J. Wakerly, Stanford University'
CUSTMDEC device ’P16L8’;

" Input pins
!CS, !RD, A0, A1, A2          pin 1, 2, 3, 4, 5;
" Output pins
!BILL, !MARY, !JOAN, !PAUL    pin 19, 18, 17, 16 istype 'com';
!ANNA, !FRED, !DAVE, !KATE    pin 15, 14, 13, 12 istype 'com';

equations
BILL = CS & RD & (!A2 & !A1 & !A0);
MARY = CS & RD & (!A2 & !A1 & !A0 # !A2 & !A1 &  A0);
KATE = CS & RD & (!A2 & !A1 &  A0 #  A2 &  A1 &  A0);
JOAN = CS & RD & (!A2 &  A1 & !A0);
PAUL = CS & RD & (!A2 &  A1 &  A0);
ANNA = CS & RD & ( A2 & !A1 & !A0);
FRED = CS & RD & ( A2 & !A1 &  A0);
DAVE = CS & RD & ( A2 &  A1 & !A0);

end CUSTMDEC
```

A PLD solution to the same problem is shown in Table 5-11. Each of the last six equations uses a single AND gate in the PLD. The ABEL compiler will also minimize the MARY equation to use just one AND gate. Once again, active-high output signals could be obtained just by changing two lines in the declaration section:

```
BILL, MARY, JOAN, PAUL       pin 19, 18, 17, 16 istype 'com';
ANNA, FRED, DAVE, KATE       pin 15, 14, 13, 12 istype 'com';
```

Another way of writing the equations is shown in Table 5-12. In most applications, this style is more clear, especially if the select inputs have numeric significance.

Table 5-12 Equivalent ABEL equations for a customized decoder.

```
ADDR = [A2,A1,A0];

equations
BILL = CS & RD & (ADDR == 0);
MARY = CS & RD & (ADDR == 0) # (ADDR == 1);
KATE = CS & RD & (ADDR == 1) # (ADDR == 7);
JOAN = CS & RD & (ADDR == 2);
PAUL = CS & RD & (ADDR == 3);
ANNA = CS & RD & (ADDR == 4);
FRED = CS & RD & (ADDR == 5);
DAVE = CS & RD & (ADDR == 6);
```

5.4.7 Decoders in VHDL

There are several ways to approach the design of decoders in VHDL. The most primitive approach would be to write a structural equivalent of a decoder logic circuit, as Table 5-13 does for the 2-to-4 binary decoder of Figure 5-32 on page 314. Of course, this mechanical conversion of an existing design into the equivalent of a netlist defeats the purpose of using VHDL in the first place.

Instead, we would like to write a program that uses VHDL to make our decoder design more understandable and maintainable. Table 5-14 shows one approach to writing code for a 3-to-8 binary decoder equivalent to the 74x138, using the dataflow style of VHDL. The address inputs `A(2 downto 0)` and the active-low decoded outputs `Y_L(0 to 7)` are declared using vectors to improve readability. A `select` statement enumerates the eight decoding cases and assigns the appropriate active-low output pattern to an 8-bit internal signal `Y_L_i`. This value is assigned to the actual circuit output `Y_L` only if all of the enable inputs are asserted.

This design is a good start, and it works, but it does have a potential pitfall. The adjustments that handle the fact that two inputs and all the outputs are active-low happen to be buried in the final assignment statement. While it's true

Table 5-13 VHDL structural program for the decoder in Figure 5-32.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V2to4dec is
  port (I0, I1, EN: in STD_LOGIC;
        Y0, Y1, Y2, Y3: out STD_LOGIC );
end V2to4dec;

architecture V2to4dec_s of V2to4dec is
  signal NOTI0, NOTI1: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component and3 port (I0, I1, I2: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (I0,NOTI0);
  U2: inv port map (I1,NOTI1);
  U3: and3 port map (NOTI0,NOTI1,EN,Y0);
  U4: and3 port map (   I0,NOTI1,EN,Y1);
  U5: and3 port map (NOTI0,   I1,EN,Y2);
  U6: and3 port map (   I0,   I1,EN,Y3);
end V2to4dec_s;
```

Table 5-14 Dataflow-style VHDL program for a 74x138-like 3-to-8 binary decoder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x138 is
    port (G1, G2A_L, G2B_L: in STD_LOGIC;      -- enable inputs
        A: in STD_LOGIC_VECTOR (2 downto 0);   -- select inputs
        Y_L: out STD_LOGIC_VECTOR (0 to 7) );  -- decoded outputs
 end V74x138;

architecture V74x138_a of V74x138 is
  signal Y_L_i: STD_LOGIC_VECTOR (0 to 7);
begin
  with A select Y_L_i <=
    "01111111" when "000",
    "10111111" when "001",
    "11011111" when "010",
    "11101111" when "011",
    "11110111" when "100",
    "11111011" when "101",
    "11111101" when "110",
    "11111110" when "111",
    "11111111" when others;
  Y_L <= Y_L_i when (G1 and not G2A_L and not G2B_L)='1' else "11111111";
end V74x138_a;
```

Table 5-15 VHDL architecture with a maintainable approach to active-level handling.

```
architecture V74x138_b of V74x138 is
  signal G2A, G2B: STD_LOGIC;             -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);    -- active-high version of outputs
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);  -- internal signal
begin
  G2A <= not G2A_L; -- convert inputs
  G2B <= not G2B_L; -- convert inputs
  Y_L <= Y;         -- convert outputs
  with A select Y_s <=
    "10000000" when "000",
    "01000000" when "001",
    "00100000" when "010",
    "00010000" when "011",
    "00001000" when "100",
    "00000100" when "101",
    "00000010" when "110",
    "00000001" when "111",
    "00000000" when others;
  Y <= not Y_s when (G1 and G2A and G2B)='1' else "00000000";
end V74x138_b;
```

that most VHDL programs are written almost entirely with active-high signals, if we're defining a device with active-low external pins, we really should handle them in a more systematic and easily maintainable way.

Table 5-15 shows such an approach. No changes are made to the `entity` declarations. However, active-high versions of the active-low external pins are defined within the `V74x138_a` architecture, and explicit assignment statements are used to convert between the active-high and active-low signals. The decoder function itself is defined in terms of only the active-high signals, probably the biggest advantage of this approach. Another advantage is that the design can be easily modified in just a few well-defined places if changes are required in the external active levels.

OUT-OF-ORDER EXECUTION

In Table 5-15, we've grouped all three of the active-level conversion statements together at the beginning of program, even a value isn't assigned to `Y_L` until *after* a value is assigned to `Y`, later in the program. Remember that this is OK because the assignment statements in the architecture body are executed "concurrently." That is, an assignment to any signal causes all the other statements that use that signal to be re-evaluated, regardless of their position in the architecture body.

You could put the "`Y_L <= Y`" statement at the end of the body if its current position bothers you, but the program is a bit more maintainable in its present form, with all the active-level conversions together.

Table 5-16 Hierarchical definition of 74x138-like decoder with active-level handling.

```
architecture V74x138_c of V74x138 is
  signal G2A, G2B: STD_LOGIC;           -- active-high version of inputs
  signal Y: STD_LOGIC_VECTOR (0 to 7);  -- active-high version of outputs
  component V3to8dec port (G1, G2, G3: in STD_LOGIC;
                           A: in STD_LOGIC_VECTOR (2 downto 0);
                           Y: out STD_LOGIC_VECTOR (0 to 7) ); end component;
begin
  G2A <= not G2A_L;    -- convert inputs
  G2B <= not G2B_L;    -- convert inputs
  Y_L <= not Y;        -- convert outputs
  U1: V3to8dec port map (G1, G2A, G2B, A, Y);
end V74x138_c;
```

Active levels can be handled in an even more structured way. As shown in Table 5-16, the V74x138 architecture can be defined hierarchically, using a fully active-high V3to8dec component that has its own dataflow-style definition in Table 5-17. Once again, no changes are required in the top-level definition of the V74x138 entity. Figure 5-42 shows the relationship between the entities.

Still another approach to decoder design is shown in Table 5-18, which can replace the V3to8dec_a architecture of Table 5-17. Instead of concurrent state-

Table 5-17 Dataflow definition of an active-high 3-to-8 decoder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V3to8dec is
    port (G1, G2, G3: in STD_LOGIC;
          A: in STD_LOGIC_VECTOR (2 downto 0);
          Y: out STD_LOGIC_VECTOR (0 to 7) );
end V3to8dec;

architecture V3to8dec_a of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
    with A select Y_s <=
      "10000000" when "000",
      "01000000" when "001",
      "00100000" when "010",
      "00010000" when "011",
      "00001000" when "100",
      "00000100" when "101",
      "00000010" when "110",
      "00000001" when "111",
      "00000000" when others;
    Y <= Y_s when (G1 and G2 and G3)='1'
             else "00000000";
end V3to8dec_a;
```

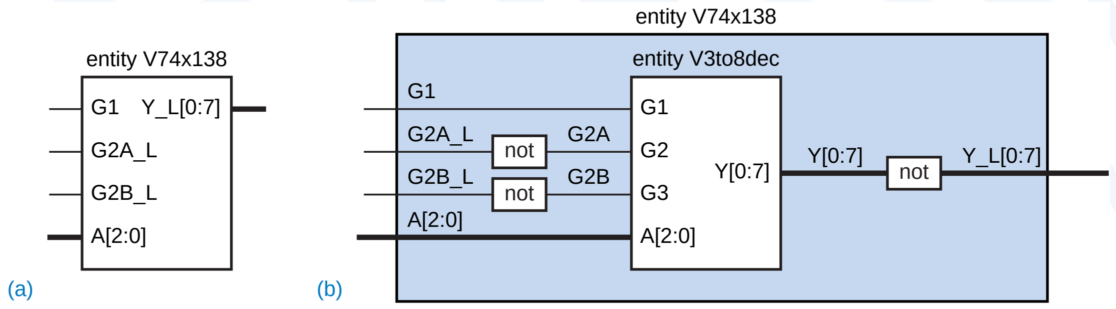


Figure 5-42 VHDL entity V74x138: (a) top level; (b) internal structure using architecture V74x138_c.

NAME MATCHING In Figure 5-42, the port names of an entity are drawn inside the corresponding box. The names of the signals that are connected to the ports when the entity is used are drawn on the signal lines. Notice that the signal names may match, but they don't have to. The VHDL compiler keeps everything straight, associating a scope with each name. The situation is completely analogous to the way variable and parameter names are handled in structured, procedural programming languages like C.

ments, this architecture uses a process and sequential statements to define the decoder's operation in a behavioral style. However, a close comparison of the two architectures shows that they're really not that different except for syntax.

Table 5-18 Behavioral-style architecture definition for a 3-to-8 decoder.

```
architecture V3to8dec_b of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR (0 to 7);
begin
process(A, G1, G2, G3, Y_s)
  begin
    case A is
      when "000" => Y_s <= "10000000";
      when "001" => Y_s <= "01000000";
      when "010" => Y_s <= "00100000";
      when "011" => Y_s <= "00010000";
      when "100" => Y_s <= "00001000";
      when "101" => Y_s <= "00000100";
      when "110" => Y_s <= "00000010";
      when "111" => Y_s <= "00000001";
      when others => Y_s <= "00000000";
    end case;
    if (G1 and G2 and G3)='1' then Y <= Y_s;
    else Y <= "00000000";
    end if;
  end process;
end V3to8dec_b;
```

Table 5-19 Truly behavioral architecture definition for a 3-to-8 decoder.

```
architecture V3to8dec_c of V3to8dec is
begin
process (G1, G2, G3, A)
  variable i: INTEGER range 0 to 7;
  begin
    Y <= "00000000";
    if (G1 and G2 and G3) = '1' then
      for i in 0 to 7 loop
        if i=CONV_INTEGER(A) then Y(i) <= '1'; end if;
      end loop;
    end if;
  end process;
end V3to8dec_c;
```

As a final example, a more truly behavioral, process-based architecture for the 3-to-8 decoder is shown in Table 5-19. (Recall that the CONV_INTEGER function was defined in \secref{VHDLconv}.) Of the examples we've given, this is the only one that describes the decoder function without essentially embedding a truth table in the VHDL program. In that respect, it is more flexible because it can be easily adapted to make a binary decoder of any size. In another respect, it is less flexible in that it does not have a truth table that can be easily modified to make custom decoders like the one we specified in Table 5-10 on page 325.

*5.4.8 Seven-Segment Decoders

seven-segment display

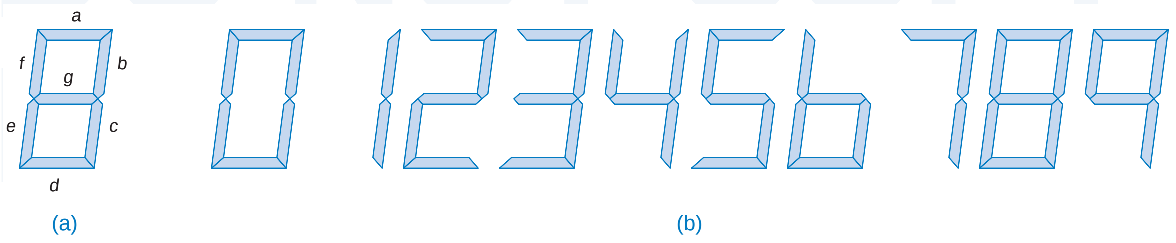
Look at your wrist and you'll probably see a *seven-segment display*. This type of display, which normally uses light-emitting diodes (LEDs) or liquid-crystal display (LCD) elements, is used in watches, calculators, and instruments to display decimal data. A digit is displayed by illuminating a subset of the seven line segments shown in Figure 5-43(a).

seven-segment decoder

74x49

A *seven-segment decoder* has 4-bit BCD as its input code and the "seven-segment code," which is graphically depicted in Figure 5-43(b), as its output code. Figure 5-44 and Table 5-20 are the logic diagram truth table and for a *74x49* seven-segment decoder. Except for the strange (clever?) connection of the "blanking input" BI_L, each output of the 74x49 is a minimal product-of-sums

Figure 5-43 Seven-segment display: (a) segment identification; (b) decimal digits.



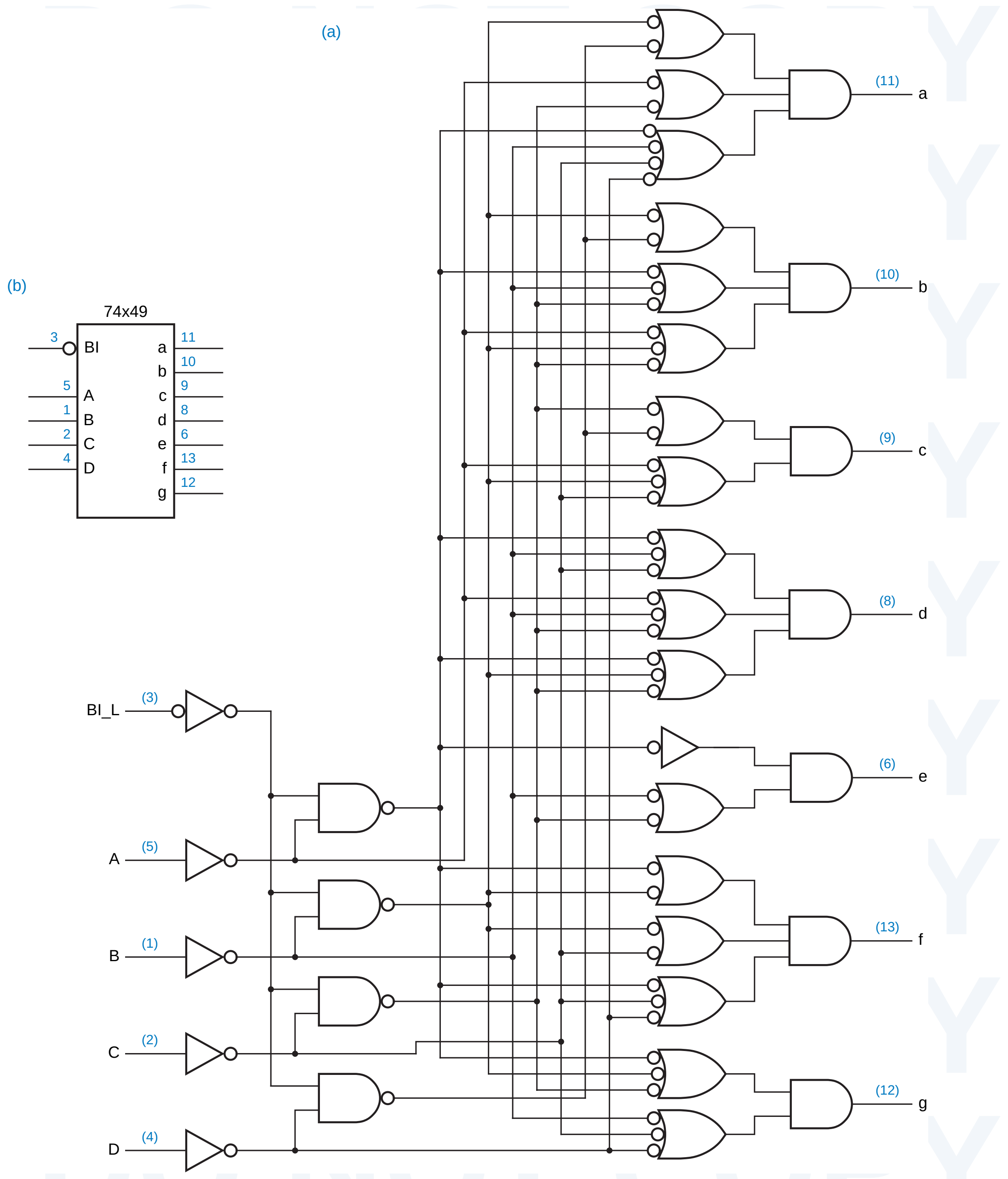


Figure 5-44 The 74x49 seven-segment decoder: (a) logic diagram, including pin numbers; (b) traditional logic symbol.

Table 5-20 Truth table for a 74x49 seven-segment decoder.

Inputs					*Outputs*						
BI_L	D	C	B	A	a	b	c	d	e	f	g
0	x	x	x	x	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	1	0	1	1	0	1	1	0	1
1	0	0	1	1	1	1	1	1	0	0	1
1	0	1	0	0	0	1	1	0	0	1	1
1	0	1	0	1	1	0	1	1	0	1	1
1	0	1	1	0	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1	0	0	0	0
1	1	0	0	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	0	0	1	1
1	1	0	1	0	0	0	0	1	1	0	1
1	1	0	1	1	0	0	1	1	0	0	1
1	1	1	0	0	0	1	0	0	0	1	1
1	1	1	0	1	1	0	0	1	0	1	1
1	1	1	1	0	0	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0	0

realization for the corresponding segment, assuming "don't-cares" for the nondecimal input combinations. The INVERT-OR-AND structure used for each output may seem a little strange, but it is equivalent under the generalized DeMorgan's theorem to an AND-OR-INVERT gate, which is a fairly fast and compact structure to build in CMOS or TTL.

Most modern seven-segment display elements have decoders built into them, so that a 4-bit BCD word can be applied directly to the device. Many of the older, discrete seven-segment decoders have special high-voltage or high-current outputs that are well suited for driving large, high-powered display elements.

Table 5-21 is an ABEL program for a seven-segment decoder. Sets are used to define the digit patterns to make the program more readable.

Table 5-21 ABEL program for a 74x49-like seven-segment decoder.

```
module Z74X49H
title 'Seven-Segment_Decoder
J. Wakerly, Micro Design Resources, Inc.'
Z74X49H device 'P16L8';

" Input pins
A, B, C, D                 pin 1, 2, 3, 4;
!BI                        pin 5;

" Output pins
SEGA, SEGB, SEGC, SEGD     pin 19, 18, 17, 16 istype 'com';
SEGE, SEGF, SEGG           pin 15, 14, 13     istype 'com';

" Set definitions
DIGITIN = [D,C,B,A];
SEGOUT = [SEGA,SEGB,SEGC,SEGD,SEGE,SEGF,SEGG];

" Segment encodings for digits
DIG0 = [1,1,1,1,1,1,0];  " 0
DIG1 = [0,1,1,0,0,0,0];  " 1
DIG2 = [1,1,0,1,1,0,1];  " 2
DIG3 = [1,1,1,1,0,0,1];  " 3
DIG4 = [0,1,1,0,0,1,1];  " 4
DIG5 = [1,0,1,1,0,1,1];  " 5
DIG6 = [1,0,1,1,1,1,1];  " 6  'tail' included
DIG7 = [1,1,1,0,0,0,0];  " 7
DIG8 = [1,1,1,1,1,1,1];  " 8
DIG9 = [1,1,1,1,0,1,1];  " 9  'tail' included
DIGA = [1,1,1,0,1,1,1];  " A
DIGB = [0,0,1,1,1,1,1];  " b
DIGC = [1,0,0,1,1,1,0];  " C
DIGD = [0,1,1,1,1,0,1];  " d
DIGE = [1,0,0,1,1,1,1];  " E
DIGF = [1,0,0,0,1,1,1];  " F

equations

SEGOUT = !BI & ( (DIGITIN ==  0) & DIG0 # (DIGITIN ==  1) & DIG1
               # (DIGITIN ==  2) & DIG2 # (DIGITIN ==  3) & DIG3
               # (DIGITIN ==  4) & DIG4 # (DIGITIN ==  5) & DIG5
               # (DIGITIN ==  6) & DIG6 # (DIGITIN ==  7) & DIG7
               # (DIGITIN ==  8) & DIG8 # (DIGITIN ==  9) & DIG9
               # (DIGITIN == 10) & DIGA # (DIGITIN == 11) & DIGB
               # (DIGITIN == 12) & DIGC # (DIGITIN == 13) & DIGD
               # (DIGITIN == 14) & DIGE # (DIGITIN == 15) & DIGF );

end Z74X49H
```

5.5 Encoders

encoder

A decoder's output code normally has more bits than its input code. If the device's output code has *fewer* bits than the input code, the device is usually called an *encoder*. For example, consider a device with eight input bits representing an unsigned binary number, and two output bits indicating whether the number is prime or divisible by 7. We might call such a device a lucky/prime encoder.

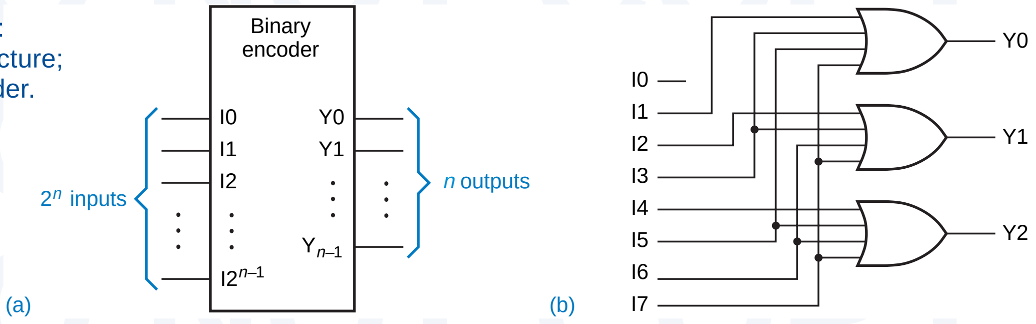
2^n-to-n encoder
binary encoder

Probably the simplest encoder to build is a 2^n-to-n or *binary encoder*. As shown in Figure 5-45(a), it has just the opposite function as a binary *de*coder—its input code is the 1-out-of-2^n code and its output code is n-bit binary. The equations for an 8-to-3 encoder with inputs I0–I7 and outputs Y0–Y2 are given below:

$$\begin{aligned}
\mathrm{Y0} &= \mathrm{I1} + \mathrm{I3} + \mathrm{I5} + \mathrm{I7}\\
\mathrm{Y1} &= \mathrm{I2} + \mathrm{I3} + \mathrm{I6} + \mathrm{I7}\\
\mathrm{Y2} &= \mathrm{I4} + \mathrm{I5} + \mathrm{I6} + \mathrm{I7}
\end{aligned}$$

The corresponding logic circuit is shown in (b). In general, a 2^n-to-n encoder can be built from n 2^{n-1}-input OR gates. Bit i of the input code is connected to OR gate j if bit j in the binary representation of i is 1.

Figure 5-45
Binary encoder:
(a) general structure;
(b) 8-to-3 encoder.



5.5.1 Priority Encoders

The 1-out-of-2^n coded outputs of an n-bit binary decoder are generally used to control a set of 2^n devices, where at most one device is supposed to be active at any time. Conversely, consider a system with 2^n *inputs*, each of which indicates a request for service, as in Figure 5-46. This structure is often found in microprocessor input/output subsystems, where the inputs might be interrupt requests.

In this situation, it may seem natural to use a binary encoder of the type shown in Figure 5-45 to observe the inputs and indicate which one is requesting service at any time. However, this encoder works properly only if the inputs are guaranteed to be asserted at most one at a time. If multiple requests can be made

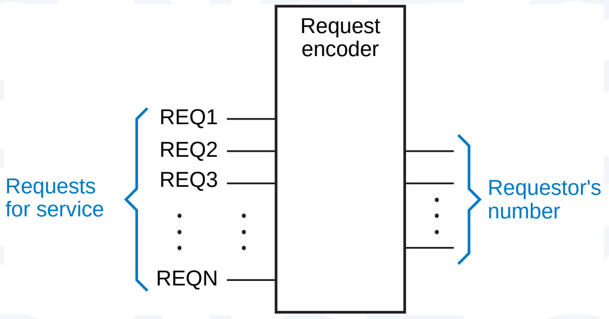


Figure 5-46 A system with 2^n requestors, and a "request encoder" that indicates which request signal is asserted at any time.

simultaneously, the encoder gives undesirable results. For example, suppose that inputs I2 and I4 of the 8-to-3 encoder are both 1; then the output is 110, the binary encoding of 6.

Either 2 or 4, not 6, would be a useful output in the preceding example, but how can the encoding device decide which? The solution is to assign *priority* to the input lines, so that when multiple requests are asserted, the encoding device produces the number of the highest-priority requestor. Such a device is called a *priority encoder.*

priority

priority encoder

The logic symbol for an 8-input priority encoder is shown in Figure 5-47. Input I7 has the highest priority. Outputs A2–A0 contain the number of the highest-priority asserted input, if any. The IDLE output is asserted if no inputs are asserted.

In order to write logic equations for the priority encoder's outputs, we first define eight intermediate variables H0–H7, such that Hi is 1 if and only if Ii is the highest priority 1 input:

$$
\begin{aligned}
H7 &= I7 \\
H6 &= I6 \cdot I7' \\
H5 &= I5 \cdot I6' \cdot I7' \\
&\dots \\
H0 &= I0 \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'
\end{aligned}
$$

Using these signals, the equations for the A2–A0 outputs are similar to the ones for a simple binary encoder:

$$
\begin{aligned}
A2 &= H4 + H5 + H6 + H7 \\
A1 &= H2 + H3 + H6 + H7 \\
A0 &= H1 + H3 + H5 + H7
\end{aligned}
$$

The IDLE output is 1 if no inputs are 1:

$$
\begin{aligned}
IDLE &= (I0 + I1 + I2 + I3 + I4 + I5 + I6 + I7)' \\
&= I0' \cdot I1' \cdot I2' \cdot I3' \cdot I4' \cdot I5' \cdot I6' \cdot I7'
\end{aligned}
$$

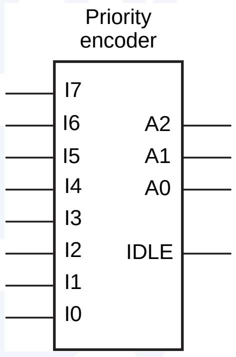


Figure 5-47 Logic symbol for a generic 8-input priority encoder.

5.5.2 The 74x148 Priority Encoder

74x148

The *74x148* is a commercially available, MSI 8-input priority encoder. Its logic symbol is shown in Figure 5-48 and its schematic is shown in Figure 5-49. The main difference between this IC and the “generic” priority encoder of Figure 5-47 is that its inputs and outputs are active low. Also, it has an enable input, EI_L, that must be asserted for any of its outputs to be asserted. The complete truth table is given in Table 5-22.

Instead of an IDLE output, the ’148 has a GS_L output that is asserted when the device is enabled and one or more of the request inputs is asserted. The manufacturer calls this “Group Select,” but it’s easier to remember as “Got Something.” The EO_L signal is an enable *output* designed to be connected to the EI_L input of another ’148 that handles lower-priority requests. /EO is asserted if EI_L is asserted but no request input is asserted; thus, a lower-priority ’148 may be enabled.

Figure 5-50 shows how four 74x148s can be connected in this way to accept 32 request inputs and produce a 5-bit output, RA4–RA0, indicating the highest-priority requestor. Since the A2–A0 outputs of at most one ’148 will be enabled at any time, the outputs of the individual ’148s can be ORed to produce RA2–RA0. Likewise, the individual GS_L outputs can be combined in a 4-to-2 encoder to produce RA4 and RA3. The RGS output is asserted if any GS output is asserted.

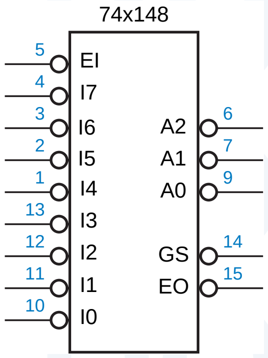


Figure 5-48 Logic symbol for the 74x148 8-input priority encoder.

Table 5-22 Truth table for a 74x148 8-input priority encoder.

Inputs									Outputs				
/EI	/I0	/I1	/I2	/I3	/I4	/I5	/I6	/I7	/A2	/A1	/A0	/GS	/EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	x	x	x	x	x	x	x	0	0	0	0	0	1
0	x	x	x	x	x	x	0	1	0	0	1	0	1
0	x	x	x	x	x	0	1	1	0	1	0	0	1
0	x	x	x	x	0	1	1	1	0	1	1	0	1
0	x	x	x	0	1	1	1	1	1	0	0	0	1
0	x	x	0	1	1	1	1	1	1	0	1	0	1
0	x	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0

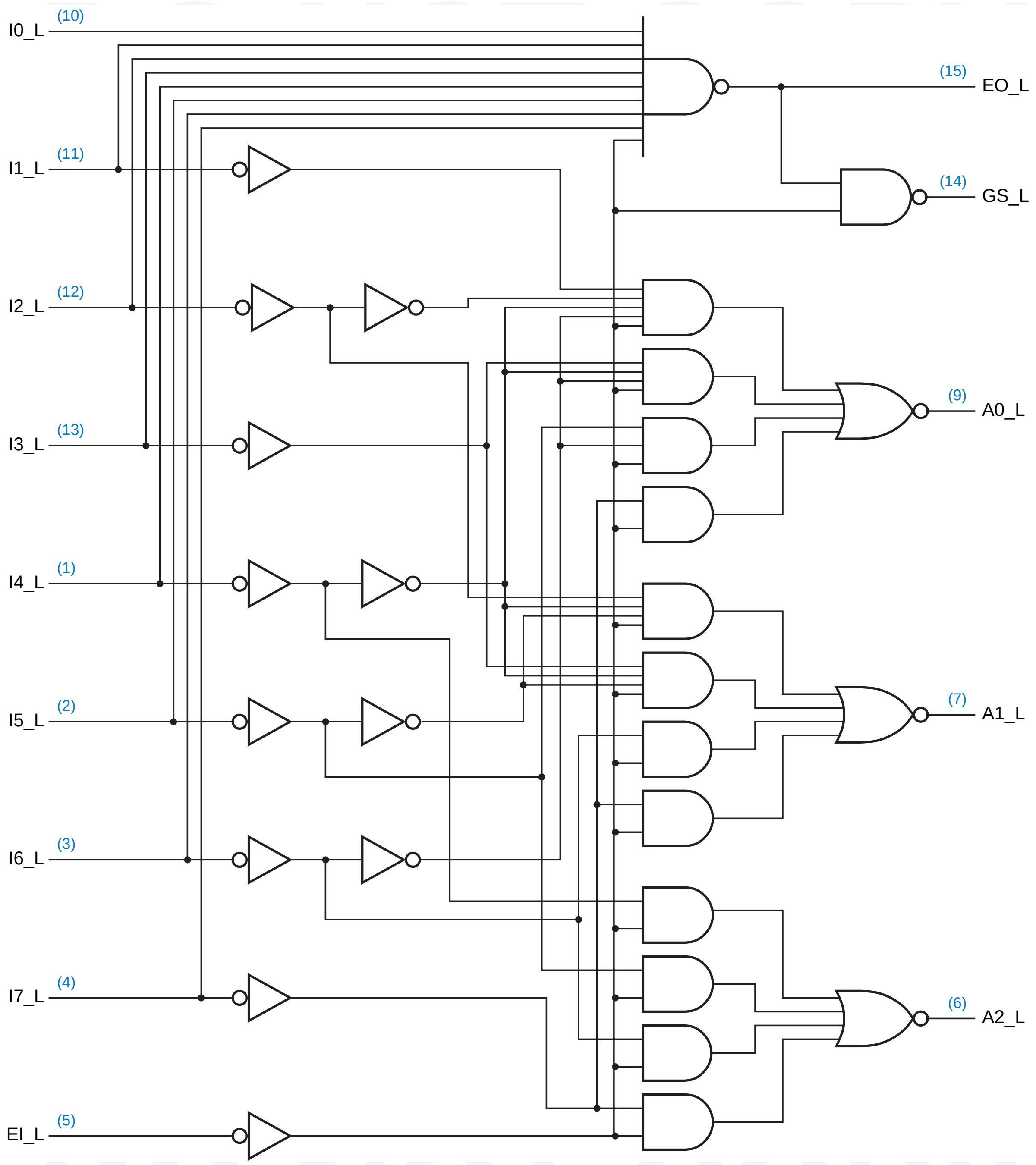


Figure 5-49 Logic diagram for the 74x148 8-input priority encoder, including pin numbers for a standard 16-pin dual in-line package.

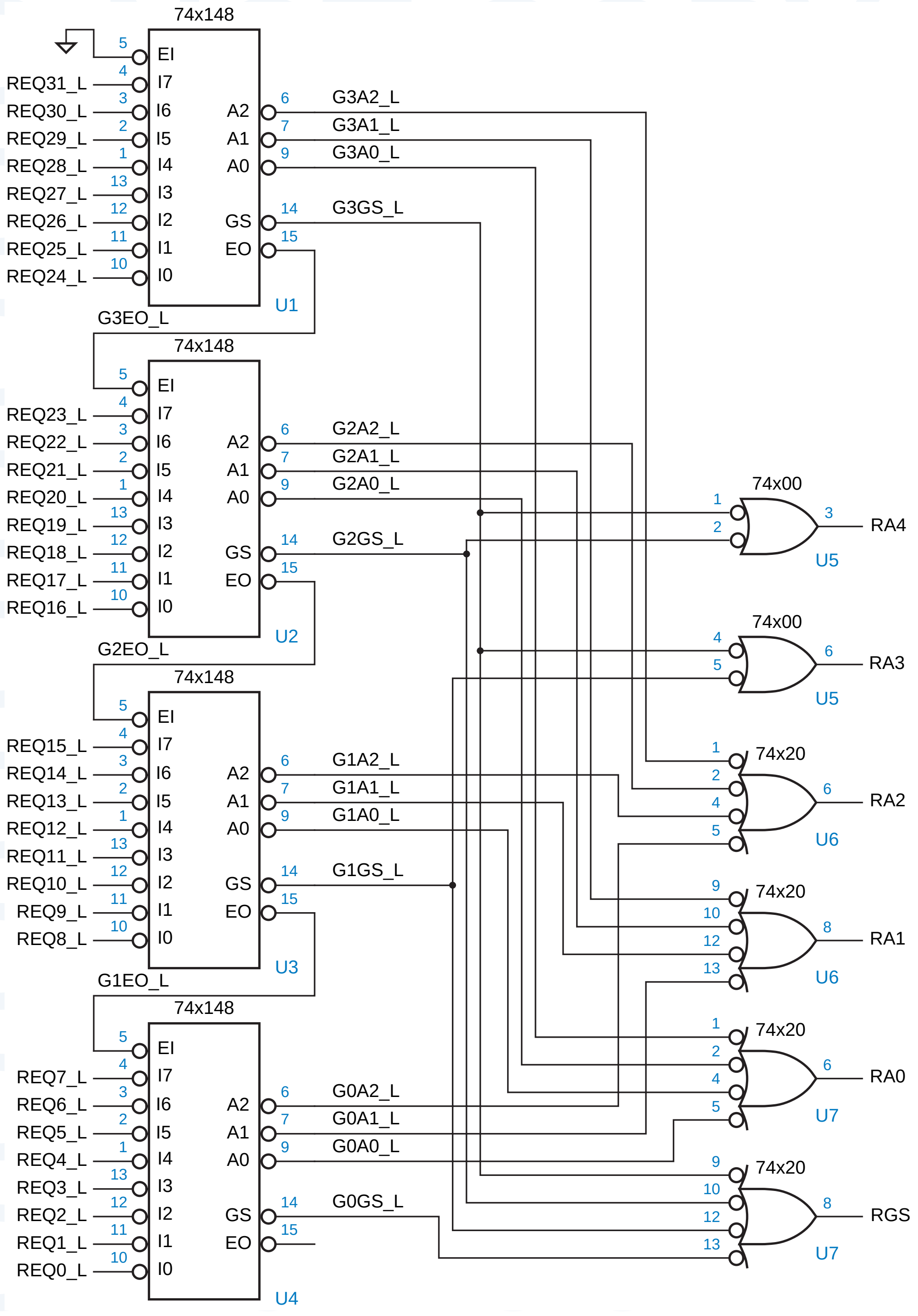


Figure 5-50 Four 74x148s cascaded to handle 32 requests.

5.5.3 Encoders in ABEL and PLDs

Encoders can be designed in ABEL using an explicit equation for each input combinations, as in Table 5-8 on page 323, or using truth tables. However, since the number of inputs is usually large, the number of input combinations is very large, and this method often is not practical.

For example, how would we specify a 15-input priority encoder for inputs P0–P14? We obviously don't want to deal with all 2^{15} possible input combinations! One way to do it is to decompose the priority function into two parts. First, we write equations for 15 variables Hi ($0 \leq i \leq 14$) such that Hi is 1 if Pi is the highest-priority asserted input. Since by definition at most one Hi variable is 1 at any time, we can combine the Hi's in a binary encoder to obtain a 4-bit number identifying the highest-priority asserted input.

An ABEL program using this approach is shown in Table 5-23, and a logic diagram for the encoder using a single PAL20L8 or GAL20V8 is given in Figure 5-51. Inputs P0–P14 are asserted to indicate requests, with P14 having the highest priority. If EN_L (Enable) is asserted, then the Y3_L–Y0_L outputs give the number (active low) of the highest-priority request, and GS is asserted if any request is present. If EN_L is negated, then the Y3_L–Y0_L outputs are negated and GS is negated. ENOUT_L is asserted if EN_L is asserted and no request is present.

Notice that in the ABEL program, the equations for the Hi variables are written as "constant expressions," before the `equations` declaration. Thus, these signals will not be generated explicitly. Rather, they will be incorporated in the subsequent equations for Y0–Y3, which the compiler cranks on to obtain

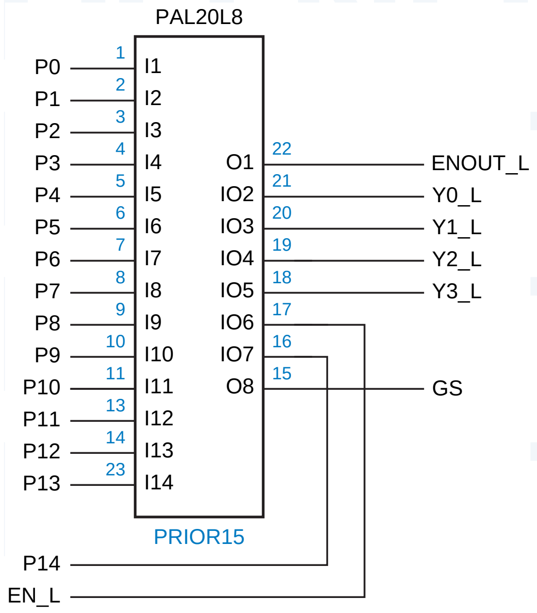


Figure 5-51
Logic diagram for a PLD-based 15-input priority encoder

Table 5-23 An ABEL program for a 15-input priority encoder.

```
module PRIOR15
title '15-Input Priority Encoder
J. Wakerly, DAVID Systems, Inc.'
PRIOR15 device 'P20L8';

" Input pins
P0, P1, P2, P3, P4, P5, P6, P7      pin 1, 2, 3, 4, 5, 6, 7, 8;
P8, P9, P10, P11, P12, P13, P14     pin 9, 10, 11, 13, 14, 23, 16;
!EN                                 pin 17;
" Output pins
!Y3, !Y2, !Y1, !Y0                  pin 18, 19, 20, 21 istype 'com';
GS, !ENOUT                          pin 15, 22         istype 'com';

" Constant expressions
H14 = EN&P14;
H13 = EN&!P14&P13;
H12 = EN&!P14&!P13&P12;
H11 = EN&!P14&!P13&!P12&P11;
H10 = EN&!P14&!P13&!P12&!P11&P10;
H9  = EN&!P14&!P13&!P12&!P11&!P10&P9;
H8  = EN&!P14&!P13&!P12&!P11&!P10&!P9&P8;
H7  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&P7;
H6  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&P6;
H5  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&P5;
H4  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&P4;
H3  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&!P4&P3;
H2  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&!P4&!P3&P2;
H1  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&!P4&!P3&!P2&P1;
H0  = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&!P4&!P3&!P2&!P1&P0;

equations
Y3 = H8 # H9 # H10 # H11 # H12 # H13 # H14;
Y2 = H4 # H5 # H6 # H7 # H12 # H13 # H14;
Y1 = H2 # H3 # H6 # H7 # H10 # H11 # H14;
Y0 = H1 # H3 # H5 # H7 # H9 # H11 # H13;

GS    = EN&(P14#P13#P12#P11#P10#P9#P8#P7#P6#P5#P4#P3#P2#P1#P0);
ENOUT = EN&!P14&!P13&!P12&!P11&!P10&!P9&!P8&!P7&!P6&!P5&!P4&!P3&!P2&!P1&!P0;

end PRIOR15
```

minimal sum-of-products expressions. As it turns out, each Yi output has only seven product terms, as you can see from the structure of the equations.

The priority encoder can be designed even more intuitively use ABEL's `WHEN` statement. As shown in Table 5-24, a deeply nested series of `WHEN` statements expresses precisely the logical function of the priority encoder. This program yields exactly the same set of output equations as the previous program.

Table 5-24 Alternate ABEL program for the same 15-input priority encoder.

```
module PRIOR15W
title '15-Input Priority Encoder'
PRIOR15W device 'P20L8';

" Input pins
P0, P1, P2, P3, P4, P5, P6, P7      pin 1, 2, 3, 4, 5, 6, 7, 8;
P8, P9, P10, P11, P12, P13, P14     pin 9, 10, 11, 13, 14, 23, 16;
!EN                                 pin 17;
" Output pins
!Y3, !Y2, !Y1, !Y0                  pin 18, 19, 20, 21 istype 'com';
GS, !ENOUT                          pin 15, 22         istype 'com';

" Sets
Y = [Y3..Y0];

equations
WHEN !EN THEN Y = 0;
ELSE WHEN P14 THEN Y = 14;
  ELSE WHEN P13 THEN Y = 13;
    ELSE WHEN P12 THEN Y = 12;
      ELSE WHEN P11 THEN Y = 11;
        ELSE WHEN P10 THEN Y = 10;
          ELSE WHEN P9 THEN Y = 9;
            ELSE WHEN P8 THEN Y = 8;
              ELSE WHEN P7 THEN Y = 7;
                ELSE WHEN P6 THEN Y = 6;
                  ELSE WHEN P5 THEN Y = 5;
                    ELSE WHEN P4 THEN Y = 4;
                      ELSE WHEN P3 THEN Y = 3;
                        ELSE WHEN P2 THEN Y = 2;
                          ELSE WHEN P1 THEN Y = 1;
                            ELSE WHEN P0 THEN Y = 0;
                              ELSE {Y = 0; ENOUT = 1;};

GS = EN&(P14#P13#P12#P11#P10#P9#P8#P7#P6#P5#P4#P3#P2#P1#P0);

end PRIOR15W
```

5.5.4 Encoders in VHDL

The approach to specifying encoders in VHDL is similar to the ABEL approach. We could embed the equivalent of a truth table or explicit equations into the VHDL program, but a behavioral description is far more intuitive. Since VHDL's IF-THEN-ELSE construct best describes prioritization and is available only within a process, we use the process-based behavioral approach.

Table 5-25 is a behavioral VHDL program for a priority encoder whose function is equivalent to the 74x148. It uses a FOR loop to look for an asserted

Table 5-25 Behavioral VHDL program for a 74x148-like 8-input priority encoder.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity V74x148 is
    port (
        EI_L: in STD_LOGIC;
        I_L: in STD_LOGIC_VECTOR (7 downto 0);
        A_L: out STD_LOGIC_VECTOR (2 downto 0);
        EO_L, GS_L: out STD_LOGIC
    );
end V74x148;

architecture V74x148p of V74x148 is
  signal EI: STD_LOGIC;                        -- active-high version of input
  signal I: STD_LOGIC_VECTOR (7 downto 0); -- active-high version of inputs
  signal EO, GS: STD_LOGIC;                    -- active-high version of outputs
  signal A: STD_LOGIC_VECTOR (2 downto 0); -- active-high version of outputs
begin
  process (EI_L, I_L, EI, EO, GS, I, A)
  variable j: INTEGER range 7 downto 0;
  begin
    EI <= not EI_L; -- convert input
    I <= not I_L;   -- convert inputs
    EO <= '1'; GS <= '0'; A <= "000";
    if (EI)='0' then EO <= '0';
    else for j in 7 downto 0 loop
        if GS = '1' then null;
        elsif I(j)='1' then
          GS <= '1'; EO <= '0'; A <= CONV_STD_LOGIC_VECTOR(j,3);
        end if;
      end loop;
    end if;
    EO_L <= not EO; -- convert output
    GS_L <= not GS; -- convert output
    A_L <= not A;   -- convert outputs
  end process;
end V74x148p;
```

input, starting with the highest-priority input. Like some of our previous programs, it performs explicit active-level conversion at the beginning and end. Also recall that the CONV_STD_LOGIC_VECTOR(j,n) function was defined in \secref{VHDLconv} to convert from an integer j to a STD_LOGIC_VECTOR of a specified length n. This program is easily modified to use a different priority order or a different number of inputs, or to add more functionality such as finding a second-highest-priority input, as explored in Exercises \exref–\exref.

5.6 Three-State Devices

In Sections 3.7.3 and 3.10.5, we described the electrical design of CMOS and TTL devices whose outputs may be in one of three states—0, 1, or Hi-Z. In this section, we'll show how to use them.

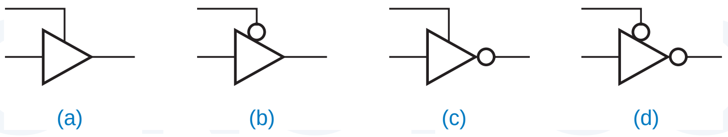
5.6.1 Three-State Buffers

three-state buffer
three-state driver
three-state enable

The most basic three-state device is a *three-state buffer*, often called a *three-state driver*. The logic symbols for four physically different three-state buffers are shown in Figure 5-52. The basic symbol is that of a noninverting buffer (a, b) or an inverter (c, d). The extra signal at the top of the symbol is a *three-state enable* input, which may be active high (a, c) or active low (b, d). When the enable input is asserted, the device behaves like an ordinary buffer or inverter. When the enable input is negated, the device output "floats"; that is, it goes to a high-impedance (Hi-Z), disconnected state and functionally behaves as if it weren't even there.

Three-state devices allow multiple sources to share a single "party line," as long as only one device "talks" on the line at a time. Figure 5-53 gives an example of how this can be done. Three input bits, SSRC2–SSRC0, select one of eight sources of data that may drive a single line, SDATA. A 3-to-8 decoder, the 74x138, ensures that only one of the eight SEL lines is asserted at a time, enabling only one three-state buffer to drive SDATA. However, if not all of the EN lines are asserted, then none of the three-state buffers is enabled. The logic value on SDATA is undefined in this case.

Figure 5-52 Various three-state buffers: (a) noninverting, active-high enable; (b) non-inverting, active-low enable; (c) inverting, active-high enable; (d) inverting, active-low enable.



DEFINING "UNDEFINED"

The actual voltage level of a floating signal depends on circuit details, such as resistive and capacitive load, and may vary over time. Also, the interpretation of this level by other circuits depends on the input characteristics of those circuits, so it's best not to count on a floating signal as being anything other than "undefined." Sometimes a pull-up resistor is used on three-state party lines to ensure that a floating value is pulled to a HIGH voltage and interpreted as logic 1. This is especially important on party lines that drive CMOS devices, which may consume excessive current when their input voltage is halfway between logic 0 and 1.

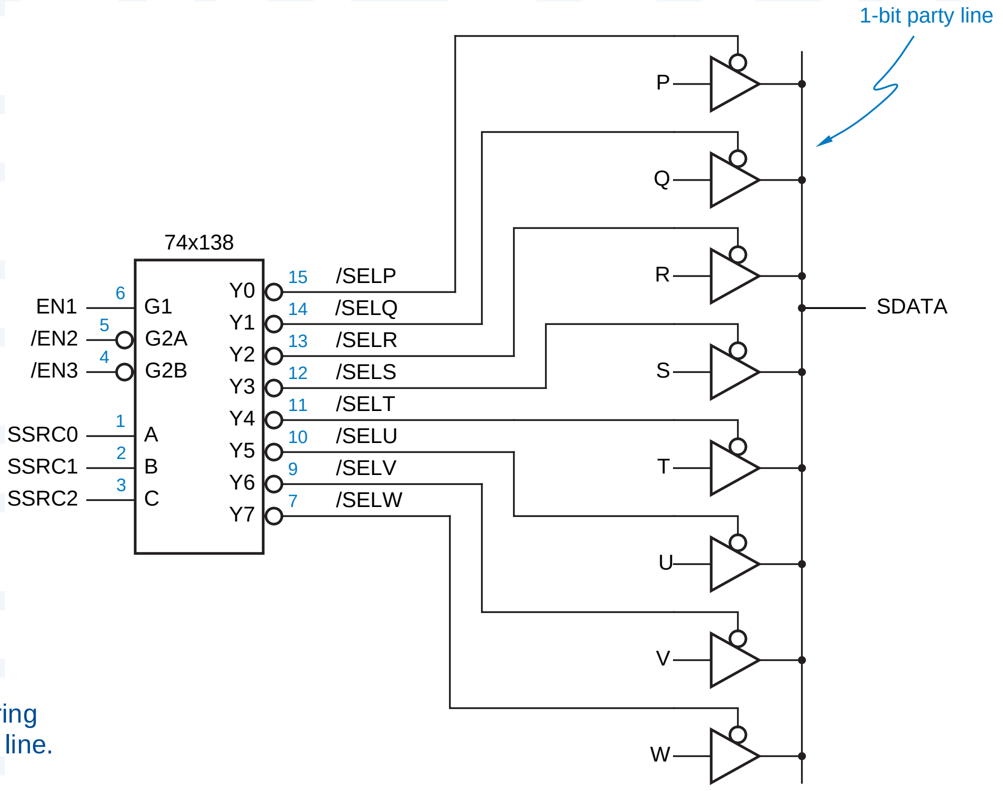


Figure 5-53
Eight sources sharing a three-state party line.

Typical three-state devices are designed so that they go into the Hi-Z state faster than they come out of the Hi-Z state. (In terms of the specifications in a data book, $t_{\rm pLZ}$ and $t_{\rm pHZ}$ are both less than $t_{\rm pZL}$ and $t_{\rm pZH}$; also see Section 3.7.3.) This means that if the outputs of two three-state devices are connected to the same party line, and we simultaneously disable one and enable the other, the first device will get off the party line before the second one gets on. This is important because, if both devices were to drive the party line at the same time, and if both were trying to maintain opposite output values (0 and 1), then excessive current would flow and create noise in the system, as discussed in Section 3.7.7. This is often called *fighting*.

fighting

Unfortunately, delays and timing skews in control circuits make it difficult to ensure that the enable inputs of different three-state devices change "simultaneously." Even when this is possible, a problem arises if three-state devices from different-speed logic families (or even different ICs manufactured on different days) are connected to the same party line. The turn-on time ($t_{\rm pZL}$ or $t_{\rm pZH}$) of a "fast" device may be shorter than the turn-off time ($t_{\rm pLZ}$ or $t_{\rm pHZ}$) of a "slow" one, and the outputs may still fight.

The only really safe way to use three-state devices is to design control logic that guarantees a *dead time* on the party line during which no one is driving it.

dead time

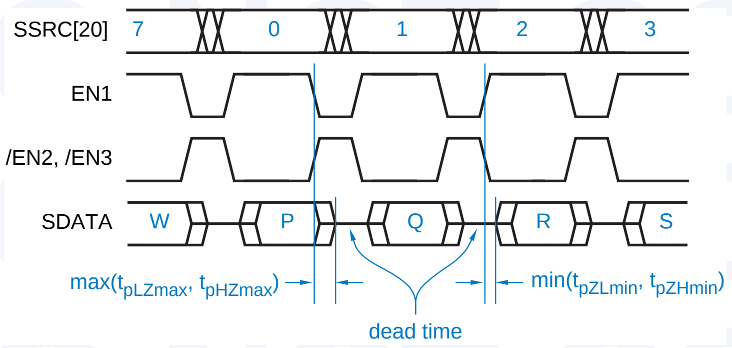

SSRC[20]
7
0
1
2
3
EN1
/EN2, /EN3
SDATA
W
P
Q
R
S
max(t_{pLZmax}, t_{pHZmax})
min(t_{pZLmin}, t_{pZHmin})
dead time

Figure 5-54 Timing diagram for the three-state party line.

The dead time must be long enough to account for the worst-case differences between turn-off and turn-on times of the devices and for skews in the three-state control signals. A timing diagram that illustrates this sort of operation for the party line of Figure 5-53 is shown in Figure 5-54. This timing diagram also illustrates a drawing convention for three-state signals—when in the Hi-Z state, they are shown at an "undefined" level halfway between 0 and 1.

5.6.2 Standard SSI and MSI Three-State Buffers

Like logic gates, several independent three-state buffers may be packaged in a single SSI IC. For example, Figure 5-55 shows the pinouts of *74x125* and *74x126*, each of which contains four independent noninverting three-state buffers in a 14-pin package. The three-state enable inputs in the '125 are active low, and in the '126 they are active high.

74x125

74x126

Most party-line applications use a bus with more than one bit of data. For example, in an 8-bit microprocessor system, the data bus is eight bits wide, and peripheral devices normally place data on the bus eight bits at a time. Thus, a peripheral device enables eight three-state drivers to drive the bus, all at the same time. Independent enable inputs, as in the '125 and '126, are not necessary.

Thus, to reduce the package size in wide-bus applications, most commonly used MSI parts contain multiple three-state buffers with common enable inputs. For example, Figure 5-56 shows the logic diagram and symbol for a *74x541* octal noninverting three-state buffer. *Octal* means that the part contains eight

74x541

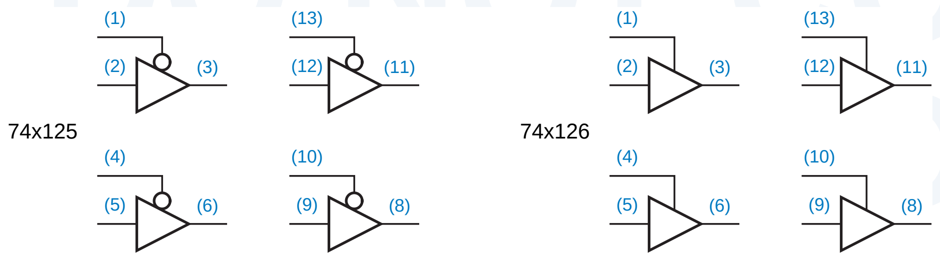


Figure 5-55 Pinouts of the 74x125 and 74x126 three-state buffers.

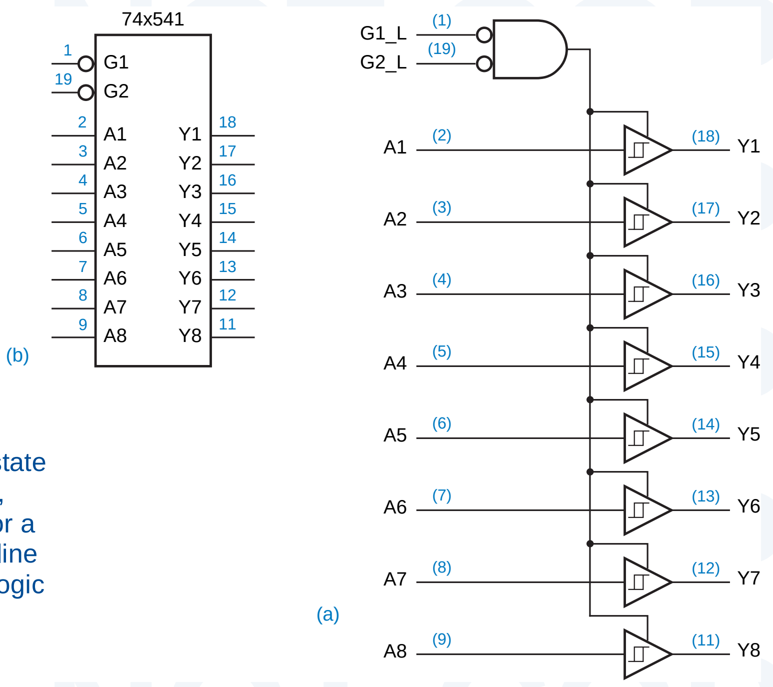


Figure 5-56
The 74x541 octal three-state buffer: (a) logic diagram, including pin numbers for a standard 20-pin dual in-line package; (b) traditional logic symbol.

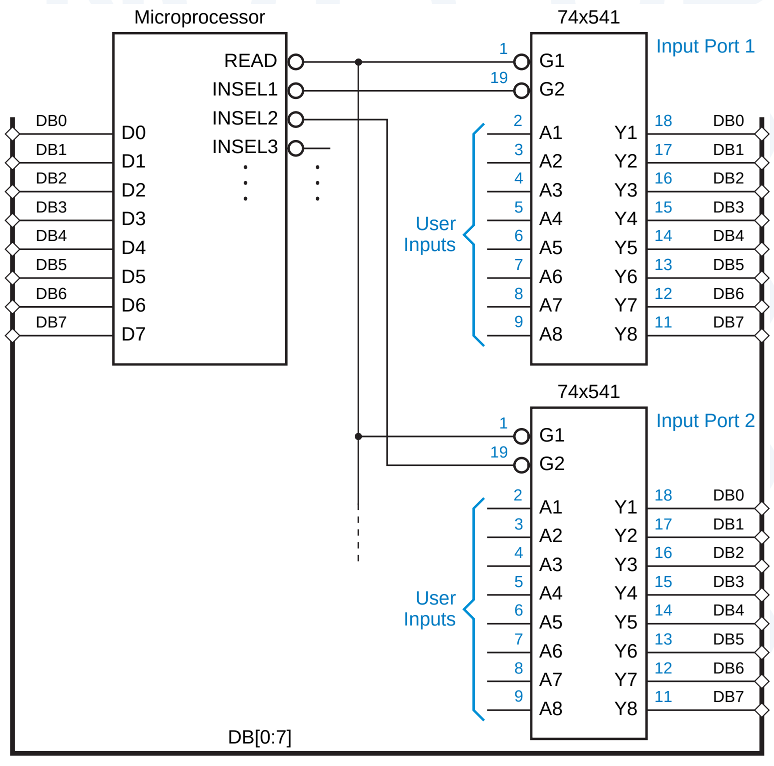
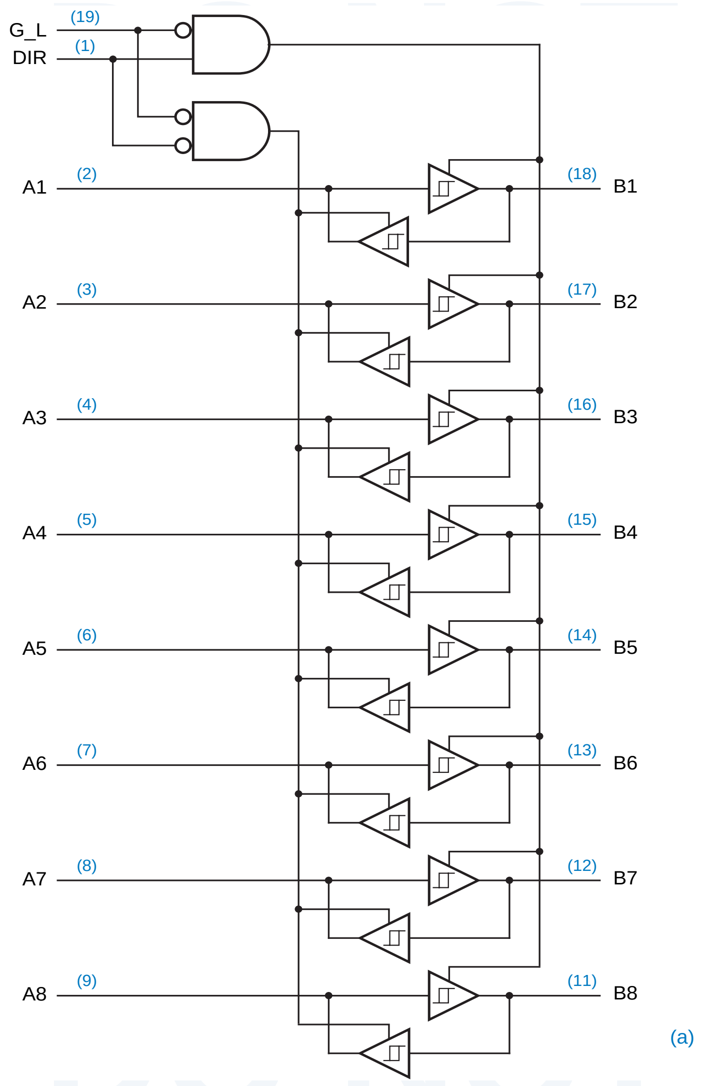


Figure 5-57
Using a 74x541 as a microprocessor input port.



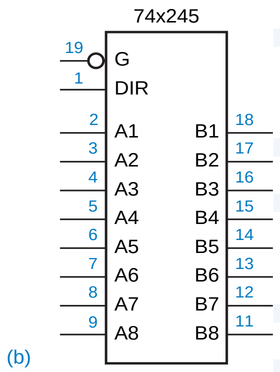


Figure 5-58
The 74x245 octal three-state transceiver: (a) logic diagram; (b) traditional logic symbol.

individual buffers. Both enable inputs, G1_L and G2_L, must be asserted to enable the device's three-state outputs. The little rectangular symbols inside the buffer symbols indicate *hysteresis*, an electrical characteristic of the inputs that improves noise immunity, as we explained in Section 3.7.2. The 74x541 inputs typically have 0.4 volts of hysteresis.

octal

hysteresis

Figure 5-57 shows part of a microprocessor system with an 8-bit data bus, DB[0–7], and a 74x541 used as an input port. The microprocessor selects Input Port 1 by asserting INSEL1 and requests a read operation by asserting READ. The selected 74x541 responds by driving the microprocessor data bus with user-supplied input data. Other input ports may be selected when a different INSEL line is asserted along with READ.

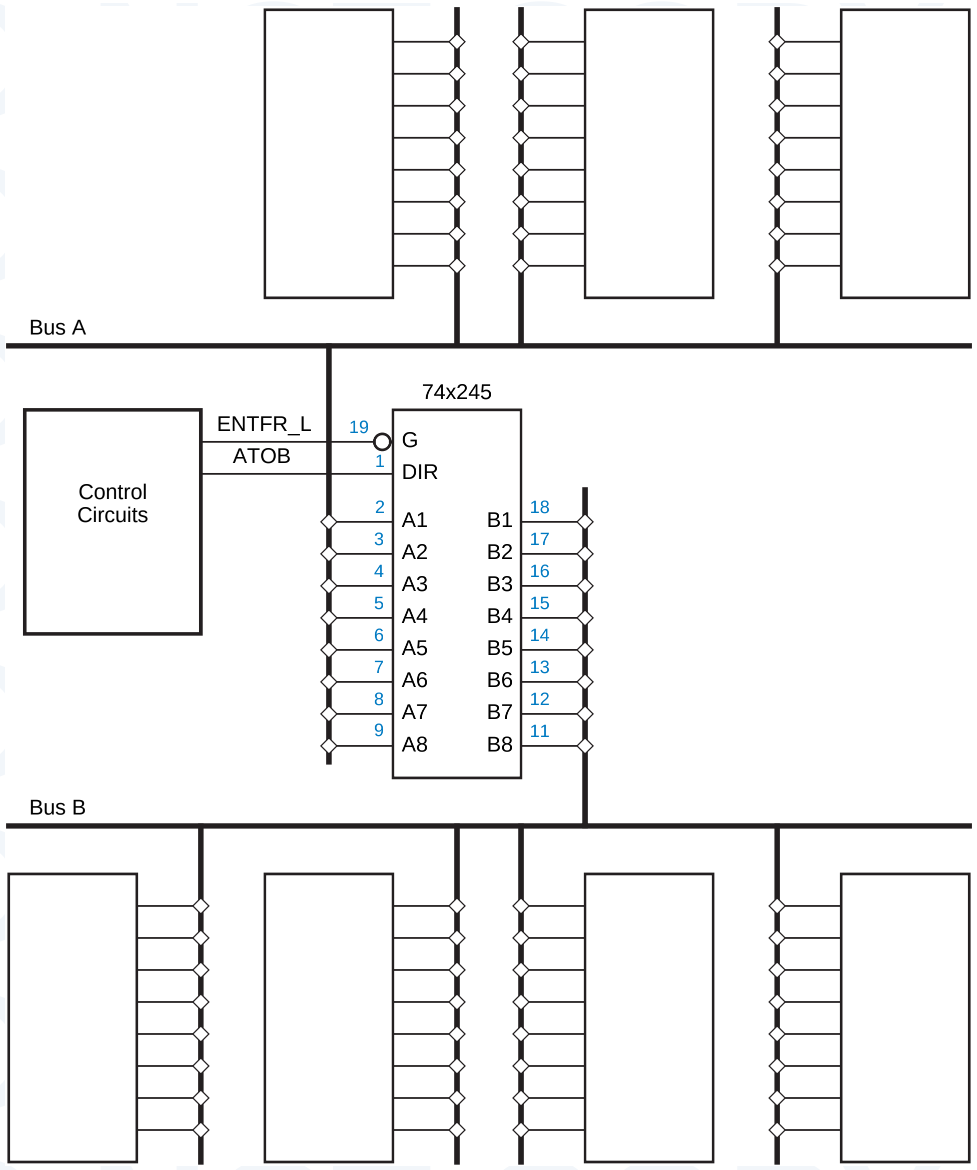


Figure 5-59
Bidirectional buses and transceiver operation.

74x540
74x240
74x241

Many other varieties of octal three-state buffers are commercially available. For example, the *74x540* is identical to the 74x541 except that it contains inverting buffers. The *74x240* and *74x241* are similar to the ’540 and ’541, except that they are split into two 4-bit sections, each with a single enable line.

bus transceiver

A *bus transceiver* contains pairs of three-state buffers connected in opposite directions between each pair of pins, so that data can be transferred in either direction. For example, Figure 5-58 on the preceding page shows the logic diagram and symbol for a *74x245* octal three-state transceiver. The DIR input

74x245

Table 5-26 Modes of operation for a pair of bidirectional buses.

ENTFR_L	ATOB	*Operation*
0	0	Transfer data from a source on bus B to a destination on bus A.
0	1	Transfer data from a source on bus A to a destination on bus B.
1	x	Transfer data on buses A and B independently.

determines the direction of transfer, from A to B (DIR = 1) or from B to A (DIR = 0). The three-state buffer for the selected direction is enabled only if G_L is asserted.

bidirectional bus

A bus transceiver is typically used between two *bidirectional buses*, as shown in Figure 5-59. Three different modes of operation are possible, depending on the state of G_L and DIR, as shown in Table 5-26. As usual, it is the designer's responsibility to ensure that neither bus is ever driven simultaneously by two devices. However, independent transfers where both buses are driven at the same time may occur when the transceiver is disabled, as indicated in the last row of the table.

5.6.3 Three-State Outputs in ABEL and PLDs

The combinational-PLD applications in previous sections have used the bidirectional I/O pins (IO2–IO7 on a PAL16L8 or GAL16V8) statically, that is, always output-enabled or always output-disabled. In such applications, the compiler can take care of programming the output-enable gates appropriately—all fuses blown, or all fuses intact. By default in ABEL, a three-state output pin is programmed to be always enabled if its signal name appears on the left-hand side of an equation, and always disabled otherwise.

.OE attribute suffix

Three-state output pins can also be controlled dynamically, by a single input, by a product term, or, using two-pass logic, by a more complex logic expression. In ABEL, an *attribute suffix .OE* is attached to a signal name on the left-hand side of an equation to indicate that the equation applies to the output-enable for the signal. In a PAL16L8 or GAL16V8, the output enable is controlled by a single AND gate, so the right-hand side of the enable equation must reduce to a single product term.

Table 5-27 shows a simple PLD program fragment with three-state control. Adapted from the program for a 74x138-like decoder on Table 5-8, this program includes a three-state output control OE for all eight decoder outputs. Notice that a set Y is defined to allow all eight output enables to be specified in a single equation; the .OE suffix is applied to each member of the set.

In the preceding example, the output pins Y0–Y7 are always either enabled or floating, and are used strictly as "output pins." I/O pins (IO2–IO7 in a 16L8 or 16V8) can be used as "bidirectional pins"; that is, they can be used dynamically

Table 5-27 ABEL program for a 74x138-like 3-to-8 binary decoder with three-state output control.

```
module Z74X138T
title '74x138 Decoder with Three-State Output Enable'
Z74X138T device 'P16L8';

" Input pins
A, B, C, !G2A, !G2B, G1, !OE  pin 1, 2, 3, 4, 5, 6, 7;
" Output pins
!Y0, !Y1, !Y2, !Y3            pin 19, 18, 17, 16 istype 'com';
!Y4, !Y5, !Y6, !Y7            pin 15, 14, 13, 12 istype 'com';

" Constant expression
ENB = G1 & G2A & G2B;
Y = [Y0..Y7];

equations
Y.OE = OE;
Y0 = ENB & !C & !B & !A;
...
Y7 = ENB &  C &  B &  A;

end Z74X138T
```

as inputs or outputs depending on whether the output-enable gate is producing a 0 or a 1. An example application of I/O pins is a four-way, 2-bit bus transceiver with the following specifications:

- The transceiver handles four 2-bit bidirectional buses, A[1:2], B[1:2], C[1:2], and D[1:2].
- The source of data to drive the buses is selected by three select inputs, S[2:0], according to Table 5-28. If S2 is 0, the buses are driven with a constant value, otherwise they are driven with one of the other buses. However, when the selected source is a bus, the source bus is driven with 00.

Table 5-28 Bus selection codes for a four-way bus transceiver.

S2	S1	S0	*Source selected*
0	0	0	00
0	0	1	01
0	1	0	10
0	1	1	11
1	0	0	A bus
1	0	1	B bus
1	1	0	C bus
1	1	1	D bus

Table 5-29 An ABEL program for four-way, 2-bit bus transceiver.

```
module XCVR4X2
title 'Four-way 2-bit Bus Transceiver'
XCVR4X2 device 'P16L8';

" Input pins
A1I, A2I                         pin 1, 11;
!AOE, !BOE, !COE, !DOE, !MOE     pin 2, 3, 4, 5, 6;
S0, S1, S2                       pin 7, 8, 9;
" Output and bidirectional pins
A1O, A2O                         pin 19, 12             istype 'com';
B1, B2, C1, C2, D1, D2           pin 18, 17, 16, 15, 14, 13 istype 'com';

" Set definitions
ABUSO = [A1O,A2O];
ABUSI = [A1I,A2I];
BBUS  = [B1,B2];
CBUS  = [C1,C2];
DBUS  = [D1,D2];
SEL   = [S2,S1,S0];
CONST = [S1,S0];
" Constants
SELA = [1,0,0];
SELB = [1,0,1];
SELC = [1,1,0];
SELD = [1,1,1];

equations
ABUSO.OE = AOE & MOE;
BBUS.OE = BOE & MOE;
CBUS.OE = COE & MOE;
DBUS.OE = DOE & MOE;
ABUSO = !S2&CONST # (SEL==SELB)&BBUS # (SEL==SELC)&CBUS # (SEL==SELD)&DBUS;
BBUS = !S2&CONST # (SEL==SELA)&ABUSI # (SEL==SELC)&CBUS # (SEL==SELD)&DBUS;
CBUS = !S2&CONST # (SEL==SELA)&ABUSI # (SEL==SELB)&BBUS # (SEL==SELD)&DBUS;
DBUS = !S2&CONST # (SEL==SELA)&ABUSI # (SEL==SELB)&BBUS # (SEL==SELC)&CBUS;

end XCVR4X2
```

- Each bus has its own output-enable signal, AOE_L, BOE_L, COE_L, or DOE_L. There is also a “master” output-enable signal, MOE_L. The transceiver drives a particular bus if and only if MOE_L and the output-enable signal for that bus are both asserted.

Table 5-29 is an ABEL program that performs the transceiver function. According to the enable (`.OE`) equations, each bus is output-enabled if MOE and its own OE are asserted. Each bus is driven with S1 and S0 if S2 is 0, and with

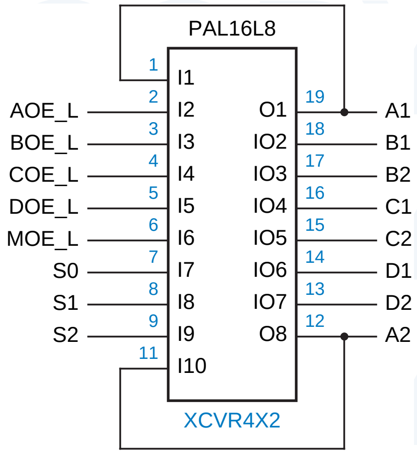


Figure 5-60 PLD inputs and outputs for a four-way, 2-bit bus transceiver.

the selected bus if a different bus is selected. If the bus itself is selected, the output equation evaluates to 0, and the bus is driven with 00 as required.

Figure 5-60 is a logic diagram for a PAL16L8 (or GAL16V8) with the required inputs and outputs. Since the device has only six bidirectional pins and the specification requires eight, the A bus uses one pair of pins for input and another for output. This is reflected in the program by the use of separate signals and sets for the A-bus input and output.

Table 5-30 IEEE 1164 package declarations for `STD_ULOGIC` and `STD_LOGIC`.

```
PACKAGE std_logic_1164 IS
-- logic state system (unresolved)
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );

-- unconstrained array of std_ulogic
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

-- resolution function
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

-- *** industry standard logic type ***
    SUBTYPE std_logic IS resolved std_ulogic;
...
```

Table 5-31 IEEE 1164 package body for `STD_ULOGIC` and `STD_LOGIC`.

```
PACKAGE BODY std_logic_1164 IS
-- local type
     TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

-- resolution function
     CONSTANT resolution_table : stdlogic_table := (
    --      ---------------------------------------------------------
    --      |  U    X    0    1    Z    W    L    H    -        |   |
    --      ---------------------------------------------------------
            ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
            ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
            ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
            ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
            ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
            ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
            ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
            ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
            ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
        );

    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
        VARIABLE result : std_ulogic := 'Z';  -- weakest state default
    BEGIN
        -- the test for a single driver is essential otherwise the
        -- loop would return 'X' for a single driver of '-' and that
        -- would conflict with the value of a single driver unresolved
        -- signal.
        IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
        ELSE
            FOR i IN s'RANGE LOOP
                result := resolution_table(result, s(i));
            END LOOP;
        END IF;
        RETURN result;
    END resolved;
...
```

*5.6.4 Three-State Outputs in VHDL

VHDL itself does not have built-in types and operators for three-state outputs. However, it does have primitives which can be used to create signals and systems with three-state behavior; the IEEE 1164 package uses these primitives. As a start, as we described in \secref{VHDL1164}, the IEEE 1164 `STD_LOGIC` type defines `'Z'` as one of its nine possible signal values; this value is used for the high-impedance state. You can assign this value to any `STD_LOGIC` signal, and the definitions of the standard logic functions account for the possibility of `'Z'` inputs (generally a `'Z'` input will cause a `'U'` output).

Given the availability of three-state signals, how do we create three-state buses in VHDL? A three-state bus generally has two or more drivers, although the mechanisms we discuss work fine with just one driver. In VHDL, there is no explicit language construct for joining three-state outputs into a bus. Instead, the compiler automatically joins together signals that are driven in two or more different processes, that is, signals that appear on the left-hand side of a signal assignment statement in two or more processes. However, the signals must have the appropriate type, as explained below.

subtype
STD_ULOGIC
unresolved type
resolution function

The IEEE 1164 STD_LOGIC type is actually defined as a *subtype* of an unresolved type, *STD_ULOGIC*. In VHDL, an *unresolved type* is used for any signal that may be driven in two or more processes. The definition of an unresolved type includes a *resolution function* that is called every time an assignment is made to a signal having that type. As the name implies, this function resolves the value of the signal when it has multiple drivers.

Tables 5-30 and 5-31 show the IEEE 1164 definitions of STD_ULOGIC, STD_LOGIC and the resolution function "resolved". This code uses a two-dimensional array resolution_table to determine the final STD_LOGIC value produced by *n* processes that drive a signal to *n* possibly different values passed in the input vector s. If, for example, a signal has four drivers, the VHDL compiler automatically constructs a 4-element vector containing the four driven values, and passes this vector to resolved every time that any one of those values changes. The result is passed back to the simulation.

Notice that the order in which the driven signal values appear in s does not affect the result produced by resolved, due to the strong ordering of "strengths" in the resolution_table: 'U'>'X'>'0,1'>'W'>'L,H'>'-'. That is, once a signal is partially resolved to a particular value, it never further resolves to a "weaker" value; and 0/1 and L/H conflicts always resolve to a stronger undefined value ('X' or 'W').

So, do you need to know all of this in order to use three-state outputs in VHDL? Well, usually not, but it can help if your simulations don't match up with reality. All that's normally required to use three-state outputs within VHDL is to declare the corresponding signals as type STD_ULOGIC.

For example, Table 5-32 describes a system that uses four 8-bit three-state drivers (in four processes) to select one of four 8-bit buses, A, B, C, and D, to drive onto a result bus X. Within each process, the IEEE 1164 standard function To_StdULogicVector is used to convert the input type of STD_LOGIC_VECTOR to STD_ULOGIC_VECTOR as required to make a legal assignment to result bus X.

VHDL is flexible enough that you can use it to define other types of bus operation. For example, you could define a subtype and resolution function for open-drain outputs such that a wired-AND function is obtained. However, the definitions for specific output types in PLDs, FPGAs, and ASICs are usually already done for you in libraries provided by the component vendors.

Table 5-32 VHDL program with four 8-bit three-state drivers.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V3statex is
    port (
        G_L: in STD_LOGIC;                          -- Global output enable
        SEL: in STD_LOGIC_VECTOR (1 downto 0);      -- Input select 0,1,2,3 ==> A,B,C,D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Input buses
        X: out STD_ULOGIC_VECTOR (1 to 8)           -- Output bus (three-state)
    );
end V3statex;

architecture V3states of V3statex is
constant ZZZZZZZZ: STD_ULOGIC_VECTOR := ('Z','Z','Z','Z','Z','Z','Z','Z');
begin
  process (G_L, SEL, A)
  begin
    if G_L='0' and SEL = "00" then X <= To_StdULogicVector(A);
    else X <= ZZZZZZZZ;
    end if;
  end process;

  process (G_L, SEL, B)
  begin
    if G_L='0' and SEL = "01" then X <= To_StdULogicVector(B);
    else X <= ZZZZZZZZ;
    end if;
  end process;

  process (G_L, SEL, C)
  begin
    if G_L='0' and SEL = "10" then X <= To_StdULogicVector(C);
    else X <= ZZZZZZZZ;
    end if;
  end process;

  process (G_L, SEL, D)
  begin
    if G_L='0' and SEL = "11" then X <= To_StdULogicVector(D);
    else X <= ZZZZZZZZ;
    end if;
  end process;

end V3states;
```

5.7 Multiplexers

multiplexer

A *multiplexer* is a digital switch—it connects data from one of n sources to its output. Figure 5-61(a) shows the inputs and outputs of an n-input, b-bit multiplexer. There are n sources of data, each of which is b bits wide, and there are b output bits. In typical commercially available multiplexers, $n = 1, 2, 4, 8$, or 16, and $b = 1, 2$, or 4. There are s inputs that select among the n sources, so $s = \lceil \log_2 n \rceil$. An enable input EN allows the multiplexer to “do its thing”; when EN = 0, all of the outputs are 0. A multiplexer is often called a *mux* for short.

mux

Figure 5-61(b) shows a switch circuit that is roughly equivalent to the multiplexer. However, unlike a mechanical switch, a multiplexer is a unidirectional device: information flows only from inputs (on the left) to outputs (on the right).

We can write a general logic equation for a multiplexer output:

$$\mathrm{iY} = \sum_{j=0}^{n-1} \mathrm{EN} \cdot \mathrm{M}_j \cdot \mathrm{iDj}$$

Here, the summation symbol represents a logical sum of product terms. Variable iY is a particular output bit ($1 \le i \le b$), and variable iDj is input bit i of source j ($0 \le j \le n-1$). M_j represents minterm j of the s select inputs. Thus, when the multiplexer is enabled and the value on the select inputs is j, each output iY equals the corresponding bit of the selected input, iDj.

Multiplexers are obviously useful devices in any application in which data must be switched from multiple sources to a destination. A common application in computers is the multiplexer between the processor’s registers and its arithmetic logic unit (ALU). For example, consider a 16-bit processor in which

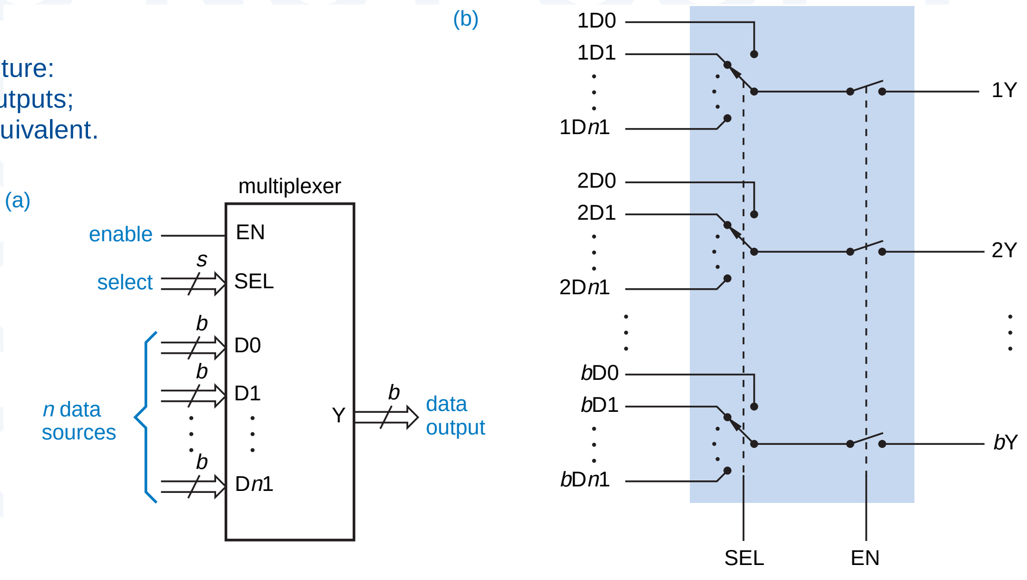


Figure 5-61
Multiplexer structure:
(a) inputs and outputs;
(b) functional equivalent.

each instruction has a 3-bit field that specifies one of eight registers to use. This 3-bit field is connected to the select inputs of an 8-input, 16-bit multiplexer. The multiplexer's data inputs are connected to the eight registers, and its data outputs are connected to the ALU to execute the instruction using the selected register.

5.7.1 Standard MSI Multiplexers

The sizes of commercially available MSI multiplexers are limited by the number of pins available in an inexpensive IC package. Commonly used muxes come in 16-pin packages. At one extreme is the *74x151*, shown in Figure 5-62, which selects among eight 1-bit inputs. The select inputs are named C, B, and A, where C is most significant numerically. The enable input EN_L is active low; both active-high (Y) and active-low (Y_L) versions of the output are provided.

74x151

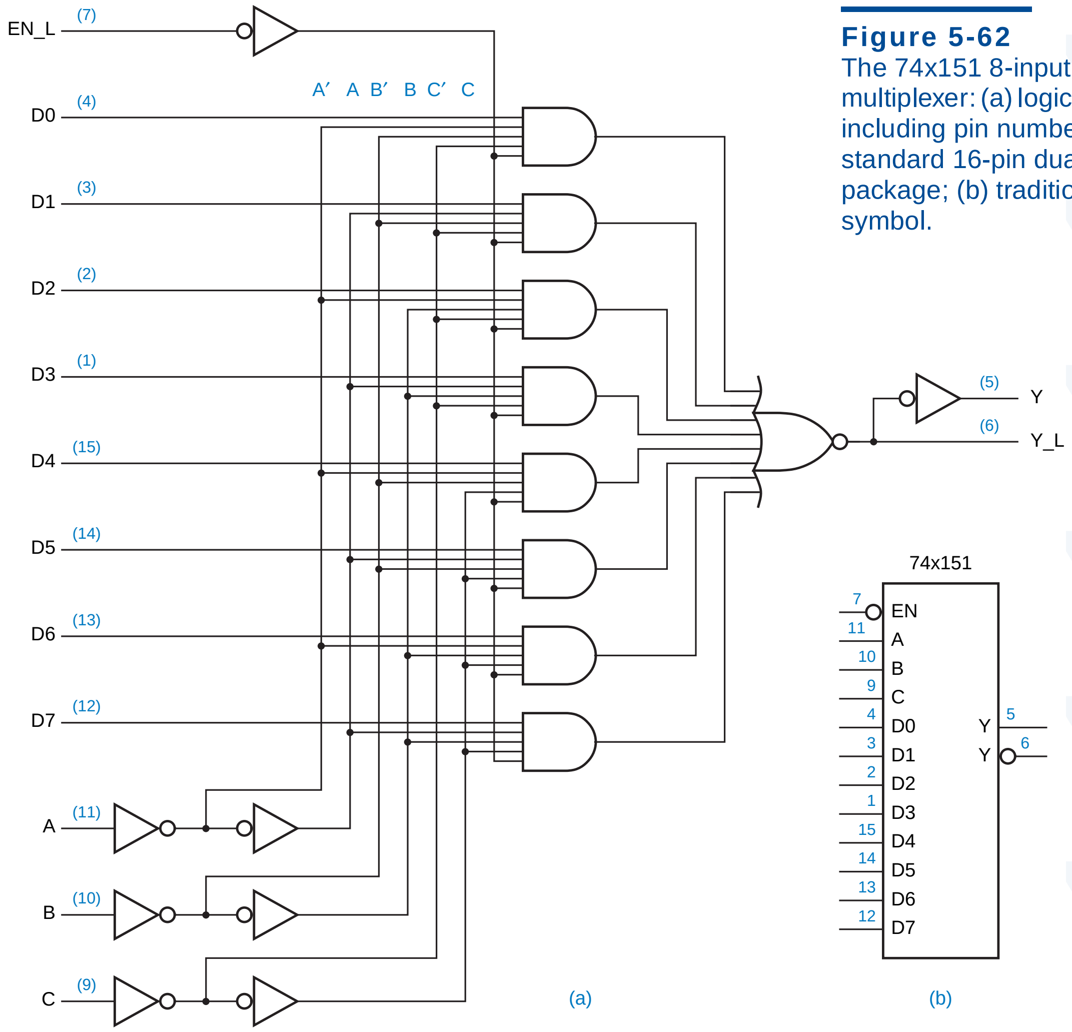


Figure 5-62
The 74x151 8-input, 1-bit multiplexer: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

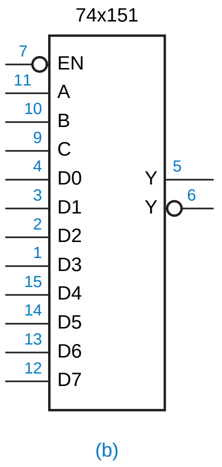
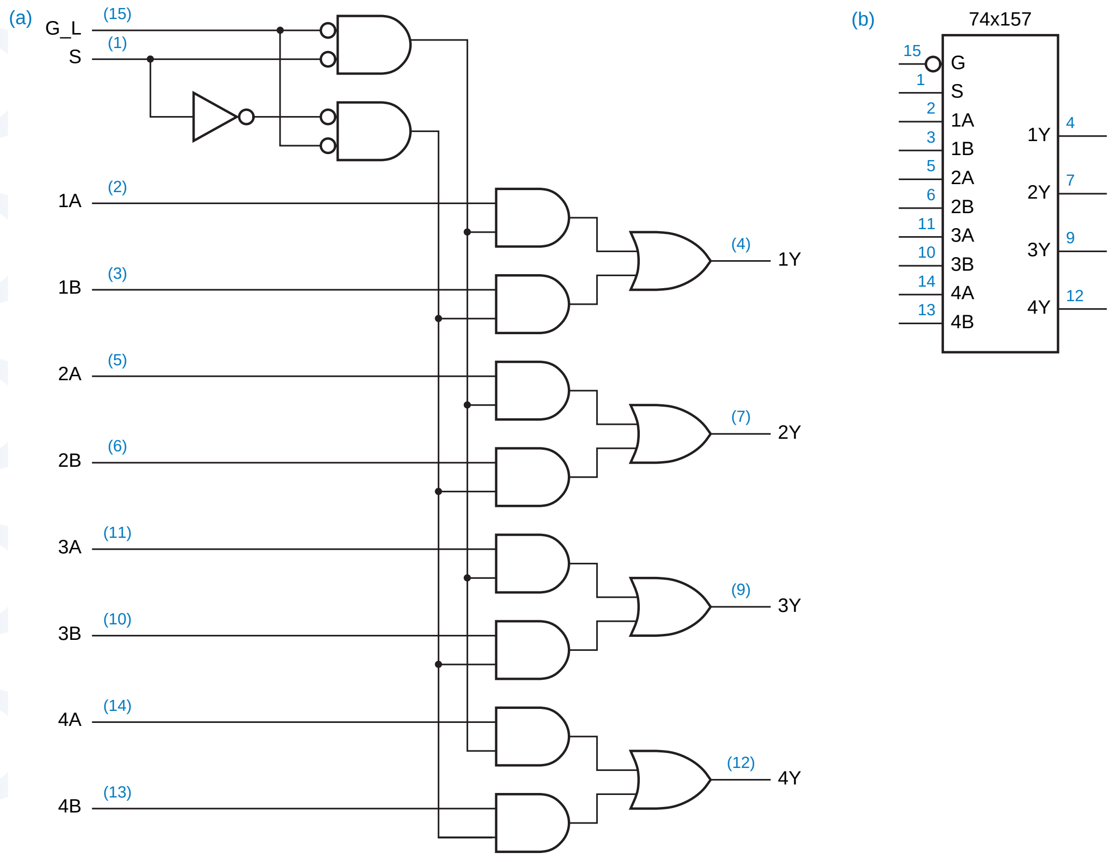


Table 5-33 Truth table for a 74x151 8-input, 1-bit multiplexer.

Inputs				*Outputs*	
EN_L	C	B	A	Y	Y_L
1	x	x	x	0	1
0	0	0	0	D0	D0′
0	0	0	1	D1	D1′
0	0	1	0	D2	D2′
0	0	1	1	D3	D3′
0	1	0	0	D4	D4′
0	1	0	1	D5	D5′
0	1	1	0	D6	D6′
0	1	1	1	D7	D7′

Figure 5-63 The 74x157 2-input, 4-bit multiplexer: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.



Inputs		*Outputs*			
G_L	S	1Y	2Y	3Y	4Y
1	x	0	0	0	0
0	0	1A	2A	3A	4A
0	1	1B	2B	3B	4B

Table 5-34 Truth table for a 74x157 2-input, 4-bit multiplexer.

The 74x151's truth table is shown in Table 5-33. Here we have once again extended our notation for truth tables. Up until now, our truth tables have specified an output of 0 or 1 for each input combination. In the 74x151's table, only a few of the inputs are listed under the "Inputs" heading. Each output is specified as 0, 1, or a simple logic function of the remaining inputs (e.g., D0 or D0′). This notation saves eight columns and eight rows in the table, and presents the logic function more clearly than a larger table would.

74x157

At the other extreme of muxes in 16-pin packages, we have the *74x157*, shown in Figure 5-63, which selects between two 4-bit inputs. Just to confuse things, the manufacturer has named the select input S and the active-low enable input G_L. Also note that the data sources are named A and B instead of D0 and D1 as in our generic example. Our extended truth-table notation makes the 74x157's description very compact, as shown in Table 5-34.

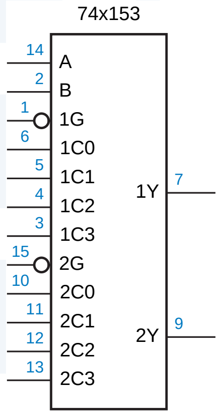
74x153

Intermediate between the 74x151 and 74x157 is the *74x153*, a 4-input, 2-bit multiplexer. This device, whose logic symbol is shown in Figure 5-64, has separate enable inputs (1G, 2G) for each bit. As shown in Table 5-35, its function is very straightforward.

Inputs				*Outputs*	
1G_L	2G_L	B	A	1Y	2Y
0	0	0	0	1C0	2C0
0	0	0	1	1C1	2C1
0	0	1	0	1C2	2C2
0	0	1	1	1C3	2C3
0	1	0	0	1C0	0
0	1	0	1	1C1	0
0	1	1	0	1C2	0
0	1	1	1	1C3	0
1	0	0	0	0	2C0
1	0	0	1	0	2C1
1	0	1	0	0	2C2
1	0	1	1	0	2C3
1	1	x	x	0	0

Table 5-35 Truth table for a 74x153 4-input, 2-bit multiplexer.

Figure 5-64 Traditional logic symbol for the 74x153.



Some multiplexers have three-state outputs. The enable input of such a multiplexer, instead of forcing the outputs to zero, forces them to the Hi-Z state. For example, the *74x251* is identical to the '151 in its pinout and its internal logic design, except that Y and Y_L are three-state outputs. When the EN_L input is negated, instead of forcing the outputs to be negated, it forces the outputs into the high-Z state. Similarly, the *74x253* and *74x257* are three-state versions of the '153 and '157. The three-state outputs are especially useful when *n*-input muxes are combined to form larger muxes, as suggested in the next subsection.

74x251

74x253

74x257

5.7.2 Expanding Multiplexers

Seldom does the size of an MSI multiplexer match the characteristics of the problem at hand. For example, we suggested earlier that an 8-input, 16-bit multiplexer might be used in the design of a computer processor. This function could be performed by 16 74x151 8-input, 1-bit multiplexers or equivalent ASIC cells, each handling one bit of all the inputs and the output. The processor's 3-bit register-select field would be connected to the A, B, and C inputs of all 16 muxes, so they would all select the same register source at any given time.

The device that produces the 3-bit register-select field in this example must have enough fanout to drive 16 loads. With 74LS-series ICs this is possible because typical devices have a fanout of 20 LS-TTL loads.

Still, it is fortunate that the '151 was designed so that each of the A, B, and C inputs presents only one LS-TTL load to the circuit driving it. Theoretically, the '151 could have been designed without the first rank of three inverters shown on the select inputs in Figure 5-62, but then each select input would have presented five LS-TTL loads, and the drivers in the register-select application would need a fanout of 80.

Another dimension in which multiplexers can be expanded is the number of data sources. For example, suppose we needed a 32-input, 1-bit multiplexer. Figure 5-65 shows one way to build it. Five select bits are required. A 2-to-4 decoder (one-half of a 74x139) decodes the two high-order select bits to enable one of four 74x151 8-input multiplexers. Since only one '151 is enabled at a time, the '151 outputs can simply be ORed to obtain the final output.

CONTROL-SIGNAL FANOUT IN ASICS

Just the sort of fanout consideration that we described above occurs quite frequently in ASIC design. When a set of control signals, such as the register-select field in the example, controls a large number of bits, the required fanout can be enormous. In CMOS chips, the consideration is not DC loading but capacitive load which slows down performance. In such an application, the designer must carefully partition the load and select points at which to buffer the control signals to reduce fanout. While inserting the extra buffers, the designer must be careful not increase the chip area significantly or to put so many buffers in series that their delay is unacceptable.

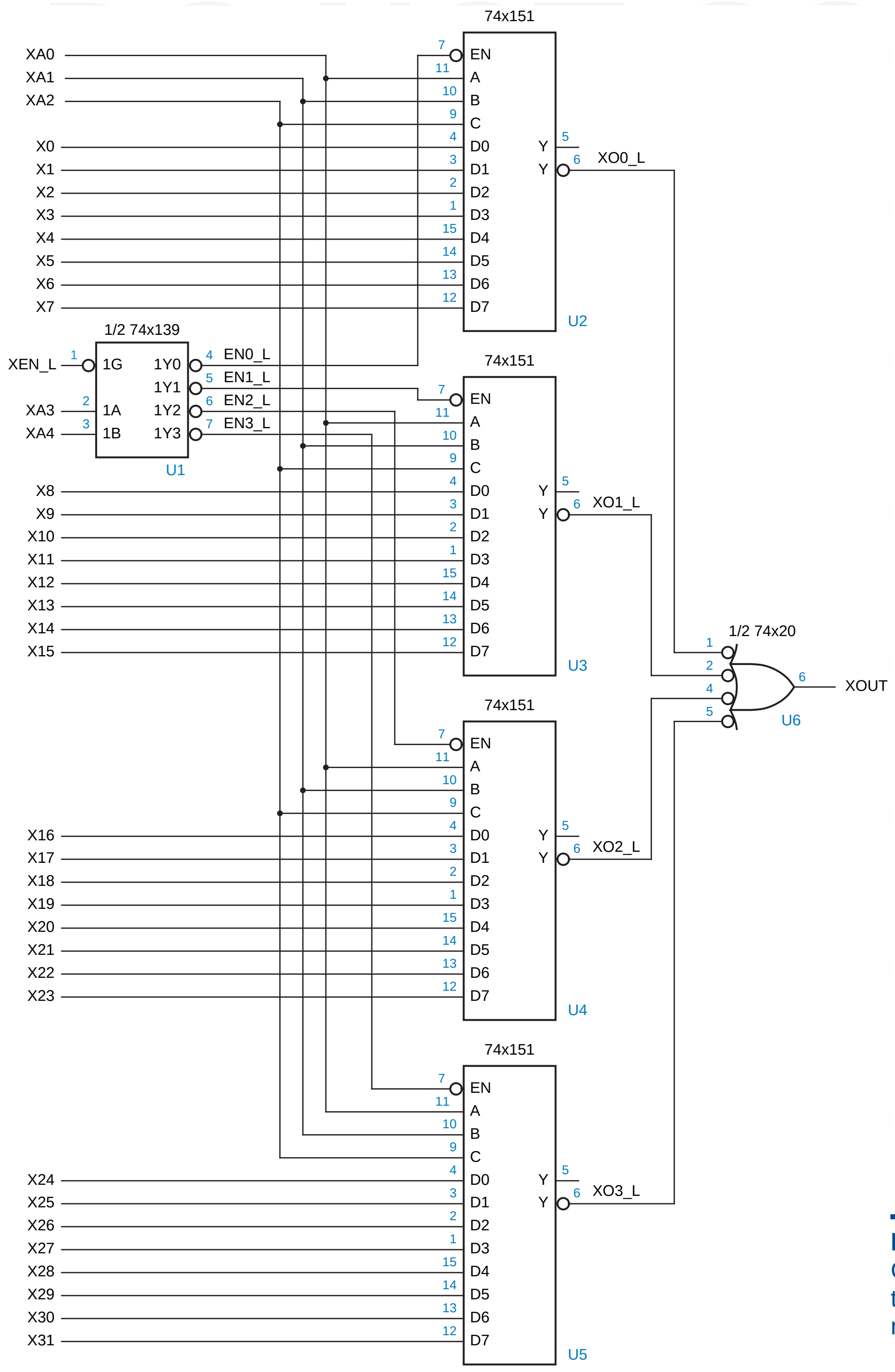


Figure 5-65 Combining 74x151s to make a 32-to-1 multiplexer.

TURN ON THE BUBBLE MACHINE

The use of bubble-to-bubble logic design should help your understanding of these multiplexer design examples. Since the decoder outputs and the multiplexer enable inputs are all active low, they can be hooked up directly. You can ignore the inversion bubbles when thinking about the logic function that is performed—you just say that when a particular decoder output is asserted, the corresponding multiplexer is enabled.

Bubble-to-bubble design also provides two options for the final OR function in Figure 5-65. The most obvious design would have used a 4-input OR gate connected to the Y outputs. However, for faster operation, we used an inverting gate, a 4-input NAND connected to the /Y outputs. This eliminated the delay of two inverters—the one used inside the ’151 to generate Y from /Y, and the extra inverter circuit that is used to obtain an OR function from a basic NOR circuit in a CMOS or TTL OR gate.

The 32-to-1 multiplexer can also be built using 74x251s. The circuit is identical to Figure 5-65, except that the output NAND gate is eliminated. Instead, the Y (and, if desired, Y_L) outputs of the four ’251s are simply tied together. The ’139 decoder ensures that at most one of the ’251s has its three-state outputs enabled at any time. If the ’139 is disabled (XEN_L is negated), then all of the ’251s are disabled, and the XOUT and XOUT_L outputs are undefined. However, if desired, resistors may be connected from each of these signals to +5 volts to pull the output HIGH in this case.

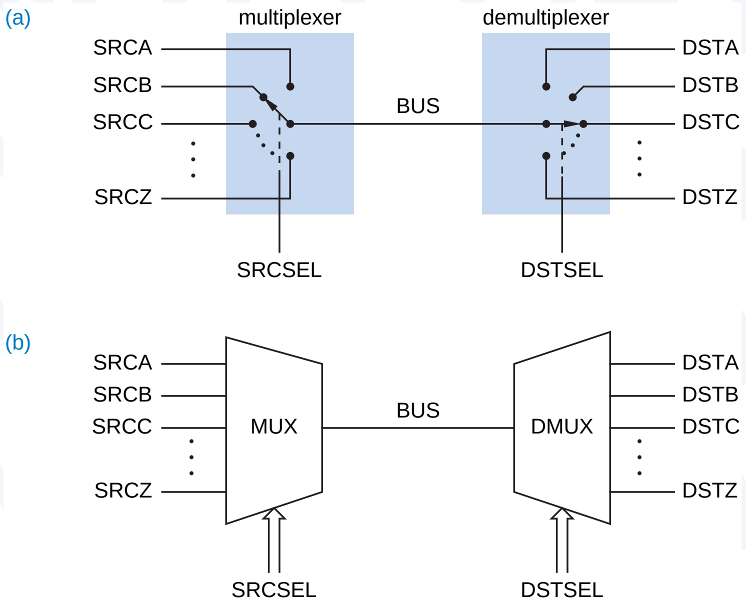


Figure 5-66 A multiplexer driving a bus and a demultiplexer receiving the bus: (a) switch equivalent; (b) block diagram symbols.

5.7.3 Multiplexers, Demultiplexers, and Buses

A multiplexer can be used to select one of *n* sources of data to transmit on a bus. At the far end of the bus, a *demultiplexer* can be used to route the bus data to one of *m* destinations. Such an application, using a 1-bit bus, is depicted in terms of our switch analogy in Figure 5-66(a). In fact, block diagrams for logic circuits often depict multiplexers and demultiplexers using the wedge-shaped symbols in (b), to suggest visually how a selected one of multiple data sources gets directed onto a bus and routed to a selected one of multiple destinations.

demultiplexer

The function of a demultiplexer is just the inverse of a multiplexer's. For example, a 1-bit, *n*-output demultiplexer has one data input and *s* inputs to select one of $n = 2^s$ data outputs. In normal operation, all outputs except the selected one are 0; the selected output equals the data input. This definition may be generalized for a *b*-bit, *n*-output demultiplexer; such a device has *b* data inputs, and its *s* select inputs choose one of $n = 2^s$ sets of *b* data outputs.

A binary decoder with an enable input can be used as a demultiplexer, as shown in Figure 5-67. The decoder's enable input is connected to the data line, and its select inputs determine which of its output lines is driven with the data bit. The remaining output lines are negated. Thus, the 74x139 can be used as a 2-bit, 4-output demultiplexer with active-low data inputs and outputs, and the 74x138 can be used as a 1-bit, 8-output demultiplexer. In fact, the manufacturer's catalog typically lists these ICs as "decoders/demultiplexers."

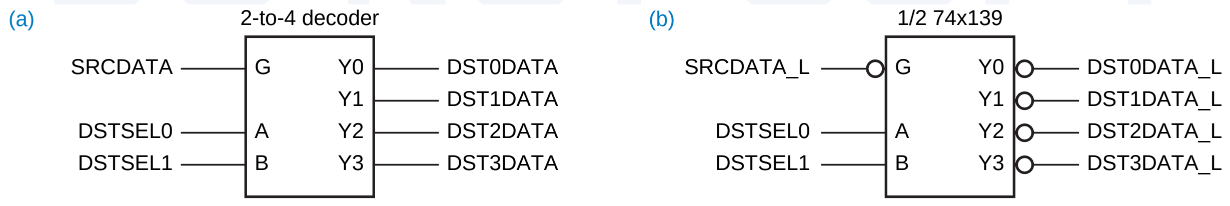


Figure 5-67 Using a 2-to-4 binary decoder as a 1-bit, 4-output demultiplexer: (a) generic decoder; (b) 74x139.

5.7.4 Multiplexers in ABEL and PLDs

Multiplexers are very easy to design using ABEL and combinational PLDs. For example, the function of a 74x153 4-input, 2-bit multiplexer can be duplicated in a PAL16L8 as shown in Figure 5-68 and Table 5-36. Several characteristics of the PLD-based design and program are worth noting:

- Signal names in the ABEL program are changed slightly from the signal names shown for a 74x153 in Figure 5-64 on page 361, since ABEL does not allow a number to be used as the first character of a signal name.
- A 74x153 has twelve inputs, while a PAL16L8 has only ten inputs. Therefore, two of the '153 inputs are assigned to 16L8 I/O pins, which are no longer usable as outputs.

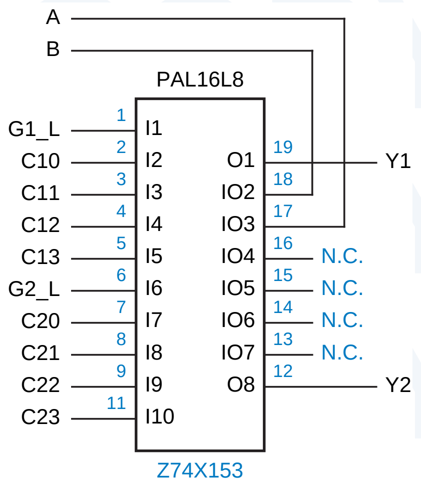


Figure 5-68
Logic diagram for the PAL16L8 used as a 74x153-like multiplexer.

- The ’153 outputs (1Y and 2Y) are assigned to pins 19 and 12 on the 16L8, which are usable *only* as outputs. This is preferable to assigning them to I/O pins; given a choice, it’s better to leave I/O pins than output-only pins as spares.
- Although the multiplexer equations in the table are written quite naturally in sum-of-products form, they don’t map directly onto the 16L8’s structure

Table 5-36
ABEL program for a 74x153-like 4-input, 2-bit multiplexer.

```
module Z74X153
title '74x153-like multiplexer PLD
J. Wakerly, Stanford University'
Z74X153 device 'P16L8';

" Input pins
A, B, !G1, !G2               pin 17, 18, 1, 6;
C10, C11, C12, C13           pin 2, 3, 4, 5;
C20, C21, C22, C23           pin 7, 8, 9, 11;
" Output pins
Y1, Y2                       pin 19, 12 istype 'com';

equations
Y1 = G1 & ( !B & !A & C10
          # !B &  A & C11
          #  B & !A & C12
          #  B &  A & C13);

Y2 = G2 & ( !B & !A & C20
          # !B &  A & C21
          #  B & !A & C22
          #  B &  A & C23);
end Z74X153
```

```
!Y1 = (!B & !A & !C10
     # !B &  A & !C11
     #  B & !A & !C12
     #  B &  A & !C13
     #  G1);
!Y2 = (!B & !A & !C20
     # !B &  A & !C21
     #  B & !A & !C22
     #  B &  A & !C23
     #  G2);
```

Table 5-37
Inverted, reduced equations for 74x153-like 4-input, 2-bit multiplexer.

because of the inverter between the AND-OR array and the actual output pins. Therefore, the ABEL compiler must complement the equations in the table and then reduce the result to sum-of-products form. With a GAL16V8, either version of the equations could be used.

Multiplexer functions are even easier to expression using ABEL's sets and relations. For example, Table 5-38 shows the ABEL program for a 4-input, 8-bit multiplexer. No `device` statement is included, because this function has too many inputs and outputs to fit in any of the PLDs we've described so far. However, it's quite obvious that a multiplexer of any size can be specified in just a few lines of code in this way.

```
module mux4in8b
title '4-input, 8-bit wide multiplexer PLD'

" Input and output pins
!G                  pin;              " Output enable for Y bus
S1..S0              pin;              " Select inputs, 0-3 ==> A-D
A1..A8, B1..B8, C1..C8, D1..D8 pin;   " 8-bit input buses A, B, C, D
Y1..Y8              pin istype 'com'; " 8-bit three-state output bus

" Sets
SEL = [S1..S0];
A = [A1..A8];
B = [B1..B8];
C = [C1..C8];
D = [D1..D8];
Y = [Y1..Y8];

equations
Y.OE = G;
WHEN (SEL == 0) THEN Y = A;
ELSE WHEN (SEL == 1) THEN Y = B;
ELSE WHEN (SEL == 2) THEN Y = C;
ELSE WHEN (SEL == 3) THEN Y = D;
end mux4in8b
```

Table 5-38
ABEL program for a 4-input, 8-bit multiplexer.

Table 5-39
Function table for a specialized 4-input, 18-bit multiplexer.

S2	S1	S0	*Input to Select*
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	B

Likewise, it is easy to customize multiplexer functions using ABEL. For example, suppose that you needed a circuit that selects one of four 18-bit input buses, A, B, C, or D, to drive a 18-bit output bus F, as specified in Table 5-39 by three control bits. There are more control-bit combinations than multiplexer inputs, so a standard 4-input multiplexer doesn't quite fit the bill (but see Exercise \exref). A 4-input, 3-bit multiplexer with the required behavior can be designed to fit into a single PAL16L8 or GAL16V8 as shown in Figure 5-69 and Table 5-40, and six copies of this device can be used to make the 18-bit mux. Alternatively, a single, larger PLD could be used. In any case, the ABEL program is very easily modified for different selection criteria.

Since this function uses all of the available pins on the PAL16L8, we had to make the pin assignment in Figure 5-69 carefully. In particular, we had to assign two output signals to the two output-only pins (O1 and O8), to maximize the number of input pins available.

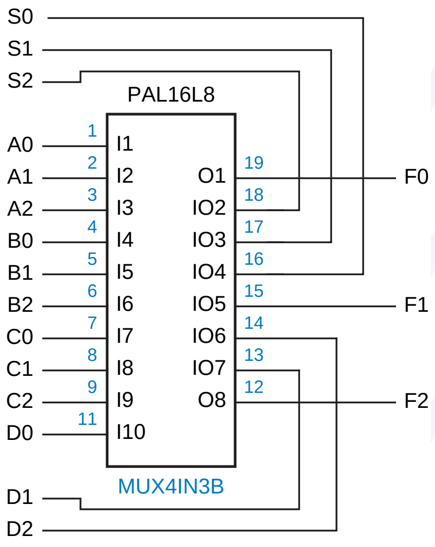


Figure 5-69
Logic diagram for the PAL16L8 used as a specialized 4-input, 3-bit multiplexer.

Table 5-40 ABEL program for a specialized 4-input, 3-bit multiplexer.

```
module mux4in3b
title 'Specialized 4-input, 3-bit Multiplexer'
mux4in3b device 'P16L8';

" Input and output pins
S2..S0                          pin 16..18;                 " Select inputs
A0..A2, B0..B2, C0..C2, D0..D2  pin 1..9, 11, 13, 14;       " Bus inputs
F0..F2                          pin 19, 15, 12 istype 'com'; " Bus outputs

" Sets
SEL = [S2..S0];
A = [A0..A2];
B = [B0..B2];
C = [C0..C2];
D = [D0..D2];
F = [F0..F2];

equations
WHEN (SEL== 0) # (SEL== 2) # (SEL== 4) # (SEL== 6) THEN F = A;
ELSE WHEN (SEL== 1) # (SEL== 7) THEN F = B;
ELSE WHEN (SEL== 3) THEN F = C;
ELSE WHEN (SEL== 5) THEN F = D;

end mux4in3b
```

EASIEST, BUT NOT CHEAPEST

As you've seen, it's very easy to program a PLD to perform decoder and multiplexer functions. Still, if you need the logic function of a standard decoder or multiplexer, it's usually less costly to use a standard MSI chip than it is to use a PLD. The PLD-based approach is best if the multiplexer has some nonstandard functional requirements, or if you think you may have to change its function as a result of debugging.

5.7.5 Multiplexers in VHDL

Multiplexers are very easy to describe in VHDL. In the dataflow style of architecture, the `SELECT` statement provides the required functionality, as shown in Table 5-41, the VHDL description of 4-input, 8-bit multiplexer.

In a behavioral architecture, a `CASE` statement is used. For example, Table 5-42 shows a process-based architecture for the same `mux4in8b` entity.

As in ABEL, it is very easy to customize the selection criteria in a VHDL multiplexer program. For example, Table 5-43 is a behavioral-style program for a specialized 4-input, 18-bit multiplexer with the selection criteria of Table 5-39.

In each example, if the select inputs are not valid (e.g., contain `U`'s or `X`'s), the output bus is set to "unknown" to help catch errors during simulation.

Table 5-41 Dataflow VHDL program for a 4-input, 8-bit multiplexer.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in8b is
    port (
        S: in STD_LOGIC_VECTOR (1 downto 0);      -- Select inputs, 0-3 ==> A-D
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 8); -- Data bus input
        Y: out STD_LOGIC_VECTOR (1 to 8)          -- Data bus output
    );
end mux4in8b;

architecture mux4in8b of mux4in8b is
begin
    with S select Y <=
      A when "00",
      B when "01",
      C when "10",
      D when "11",
      (others => 'U') when others; -- this creates an 8-bit vector of 'U'
end mux4in8b;
```

Table 5-42 Behavioral architecture for a 4-input, 8-bit multiplexer.

```
architecture mux4in8p of mux4in8b is
begin
process(S, A, B, C, D)
  begin
    case S is
      when "00" => Y <= A;
      when "01" => Y <= B;
      when "10" => Y <= C;
      when "11" => Y <= D;
      when others => Y <= (others => 'U');  -- 8-bit vector of 'U'
    end case;
  end process;
end mux4in8p;
```

5.8 EXCLUSIVE OR Gates and Parity Circuits

5.8.1 EXCLUSIVE OR and EXCLUSIVE NOR Gates

Exclusive OR (XOR)
Exclusive NOR (XNOR)
Equivalence

An *Exclusive OR (XOR)* gate is a 2-input gate whose output is 1 if exactly one of its inputs is 1. Stated another way, an XOR gate produces a 1 output if its inputs are different. An *Exclusive NOR (XNOR)* or *Equivalence* gate is just the opposite—it produces a 1 output if its inputs are the same. A truth table for these

Table 5-43 Behavioral VHDL program for a specialized 4-input, 3-bit multiplexer.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4in3b is
    port (
        S: in STD_LOGIC_VECTOR (2 downto 0);        -- Select inputs, 0-7 ==> ABACADAB
        A, B, C, D: in STD_LOGIC_VECTOR (1 to 18); -- Data bus inputs
        Y: out STD_LOGIC_VECTOR (1 to 18)          -- Data bus output
    );
end mux4in3b;

architecture mux4in3p of mux4in3b is
begin
process(S, A, B, C, D)
variable i: INTEGER;
  begin
    case S is
      when "000" | "010" | "100" | "110" => Y <= A;
      when "001" | "111" => Y <= B;
      when "011" => Y <= C;
      when "101" => Y <= D;
      when others => Y <= (others => 'U'); -- 18-bit vector of 'U'
    end case;
  end process;
end mux4in3p;
```

functions is shown in Table 5-44. The XOR operation is sometimes denoted by the symbol "⊕", that is, ⊕

$$X \oplus Y = X' \cdot Y + X \cdot Y'$$

Although EXCLUSIVE OR is not one of the basic functions of switching algebra, discrete XOR gates are fairly commonly used in practice. Most switching technologies cannot perform the XOR function directly; instead, they use multigate designs like the ones shown in Figure 5-70.

X	Y	$X \oplus Y$ (XOR)	$(X \oplus Y)'$ (XNOR)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Table 5-44 Truth table for XOR and XNOR functions.

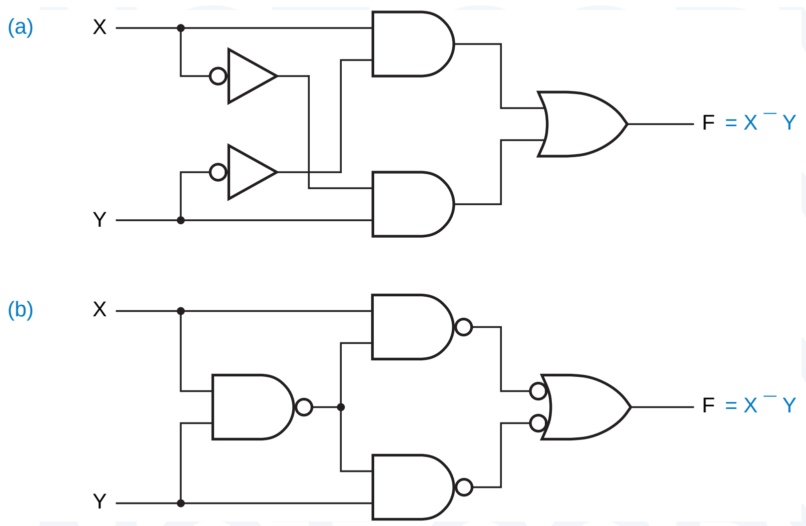


Figure 5-70 Multigate designs for the 2-input XOR function: (a) AND-OR; (b) three-level NAND.

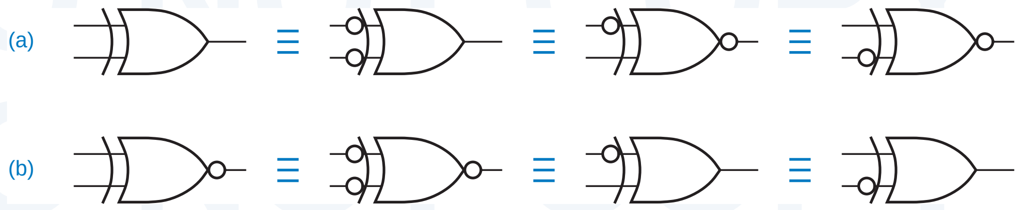


Figure 5-71 Equivalent symbols for (a) XOR gates; (b) XNOR gates.

The logic symbols for XOR and XNOR functions are shown in Figure 5-71. There are four equivalent symbols for each function. All of these alternatives are a consequence of a simple rule:

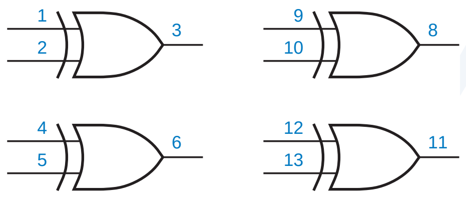
- Any two signals (inputs or output) of an XOR or XNOR gate may be complemented without changing the resulting logic function.

In bubble-to-bubble logic design, we choose the symbol that is most expressive of the logic function being performed.

74x86

Four XOR gates are provided in a single 14-pin SSI IC, the *74x86* shown in Figure 5-72. New SSI logic families do not offer XNOR gates, although they are readily available in FPGA and ASIC libraries and as primitives in HDLs.

Figure 5-72 Pinouts of the 74x86 quadruple 2-input Exclusive OR gate.



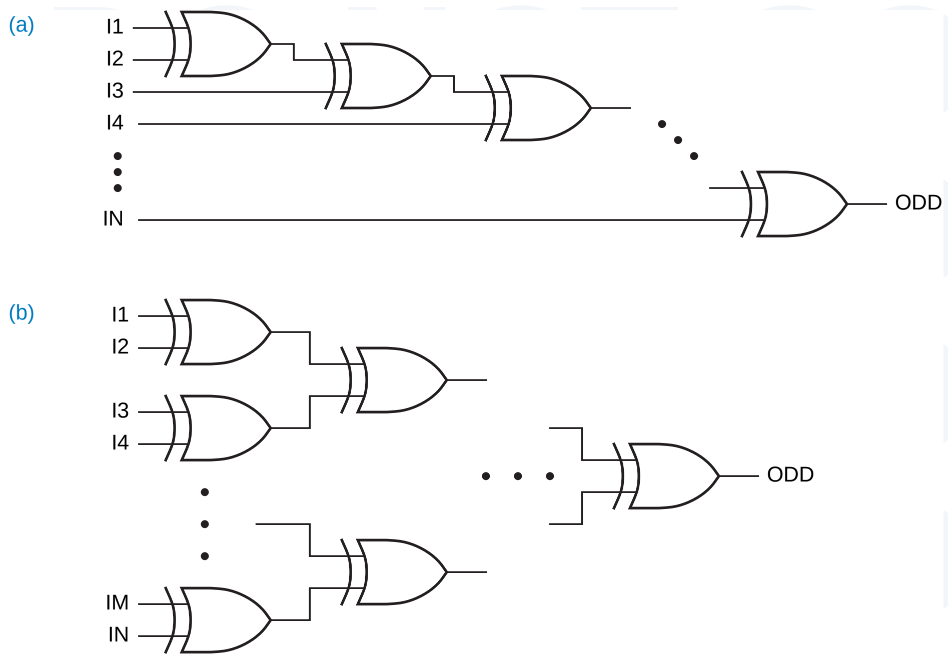


Figure 5-73 Cascading XOR gates: (a) daisy-chain connection; (b) tree structure.

5.8.2 Parity Circuits

As shown in Figure 5-73(a), n XOR gates may be cascaded to form a circuit with $n + 1$ inputs and a single output. This is called an *odd-parity circuit*, because its output is 1 if an odd number of its inputs are 1. The circuit in (b) is also an odd-parity circuit, but it's faster because its gates are arranged in a tree-like structure. If the output of either circuit is inverted, we get an *even-parity circuit*, whose output is 1 if an even number of its inputs are 1.

odd-parity circuit

even-parity circuit

5.8.3 The 74x280 9-Bit Parity Generator

Rather than build a multibit parity circuit with discrete XOR gates, it is more economical to put all of the XORs in a single MSI package with just the primary inputs and outputs available at the external pins. The *74x280* 9-bit parity generator, shown in Figure 5-74, is such a device. It has nine inputs and two outputs that indicate whether an even or odd number of inputs are 1.

74x280

5.8.4 Parity-Checking Applications

In Section 2.15, we described error-detecting codes that use an extra bit, called a parity bit, to detect errors in the transmission and storage of data. In an even-parity code, the parity bit is chosen so that the total number of 1 bits in a code word is even. Parity circuits like the 74x280 are used both to generate the correct value of the parity bit when a code word is stored or transmitted, and to check the parity bit when a code word is retrieved or received.

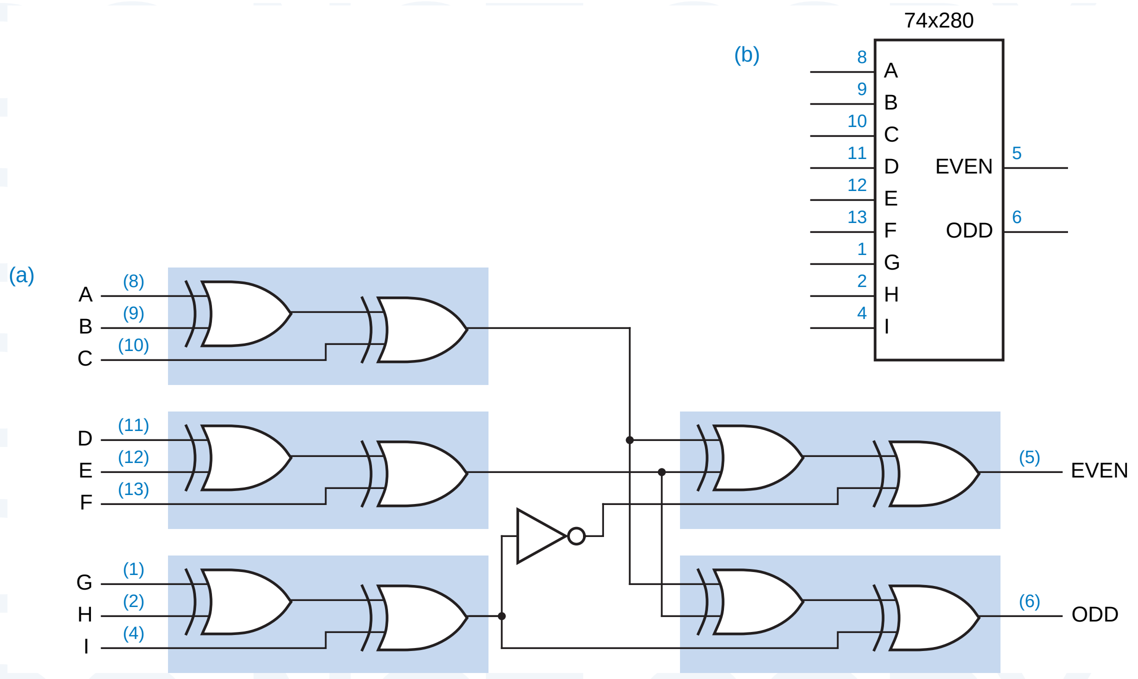


Figure 5-74 The 74x280 9-bit odd/even parity generator: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

Figure 5-75 shows how a parity circuit might be used to detect errors in the memory of a microprocessor system. The memory stores 8-bit bytes, plus a parity bit for each byte. The microprocessor uses a bidirectional bus D[0:7] to transfer data to and from the memory. Two control lines, RD and WR, are used to indicate whether a read or write operation is desired, and an ERROR signal is asserted to indicate parity errors during read operations. Complete details of the memory chips, such as addressing inputs, are not shown; memory chips are described in detail in \chapref{MEMORY}.

SPEEDING UP THE XOR TREE

If each XOR gate in Figure 5-74 were built using discrete NAND gates as in Figure 5-70(b), the 74x280 would be pretty slow, having a propagation delay equivalent to $4 \cdot 3 + 1$, or 13, NAND gates. Instead, a typical implementation of the 74x280 uses a 4-wide AND-OR-INVERT gate to perform the function of each shaded pair of XOR gates in the figure with about the same delay as a single NAND gate. The A–I inputs are buffered through two levels of inverters so that each input presents just one unit load to the circuit driving it. Thus, the total propagation delay through this implementation of the 74x280 is about the same as five inverting gates, not 13.

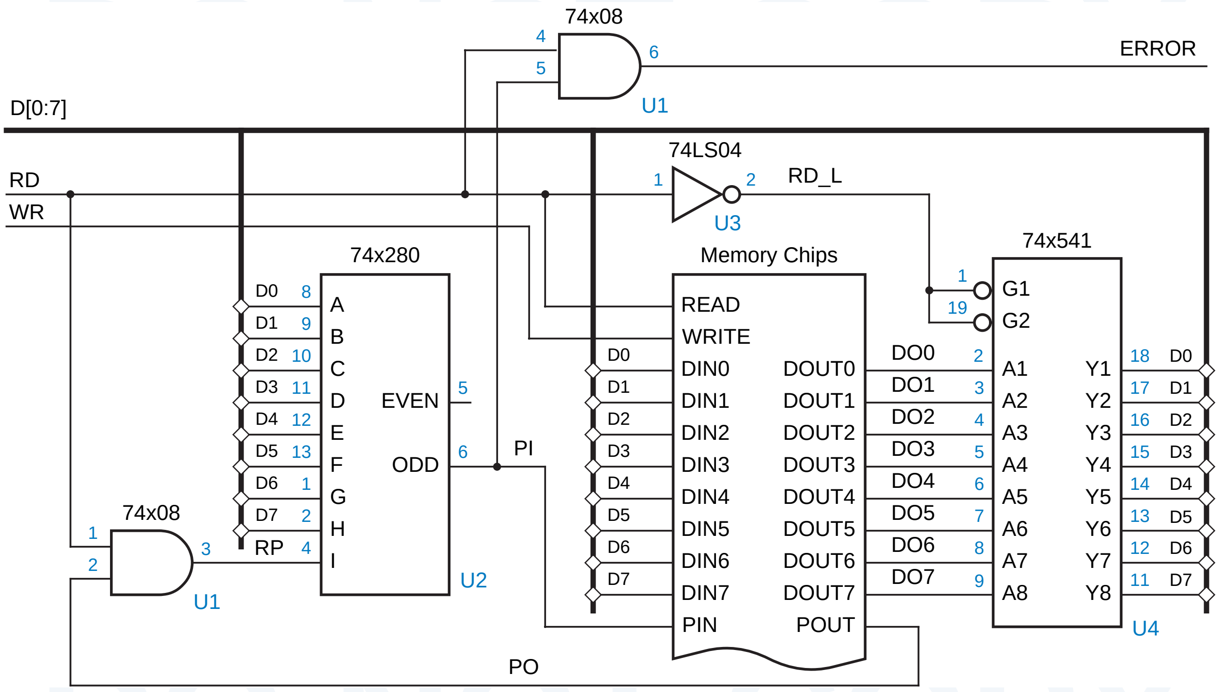


Figure 5-75 Parity generation and checking for an 8-bit-wide memory system.

To store a byte into the memory chips, we specify an address (not shown), place the byte on D[0–7], generate its parity bit on PIN, and assert WR. The AND gate on the I input of the 74x280 ensures that I is 0 except during read operations, so that during writes the ’280’s output depends only on the parity of the D-bus data. The ’280’s ODD output is connected to PIN, so that the total number of 1s stored is even.

To retrieve a byte, we specify an address (not shown) and assert RD; the byte value appears on DOUT[0–7] and its parity appears on POUT. A 74x541 drives the byte onto the D bus, and the ’280 checks its parity. If the parity of the 9-bit word DOUT[0–7],POUT is odd during a read, the ERROR signal is asserted.

Parity circuits are also used with error-correcting codes such as the Hamming codes described in Section 2.15.3. We showed the parity-check matrix for a 7-bit Hamming code in Figure 2-13 on page 59. We can correct errors in this code as shown in Figure 5-76. A 7-bit word, possibly containing an error, is presented on DU[1–7]. Three 74x280s compute the parity of the three bit-groups defined by the parity-check matrix. The outputs of the ’280s form the syndrome, which is the number of the erroneous input bit, if any. A 74x138 is used to decode the syndrome. If the syndrome is zero, the NOERROR_L signal is asserted (this signal also could be named ERROR). Otherwise, the erroneous

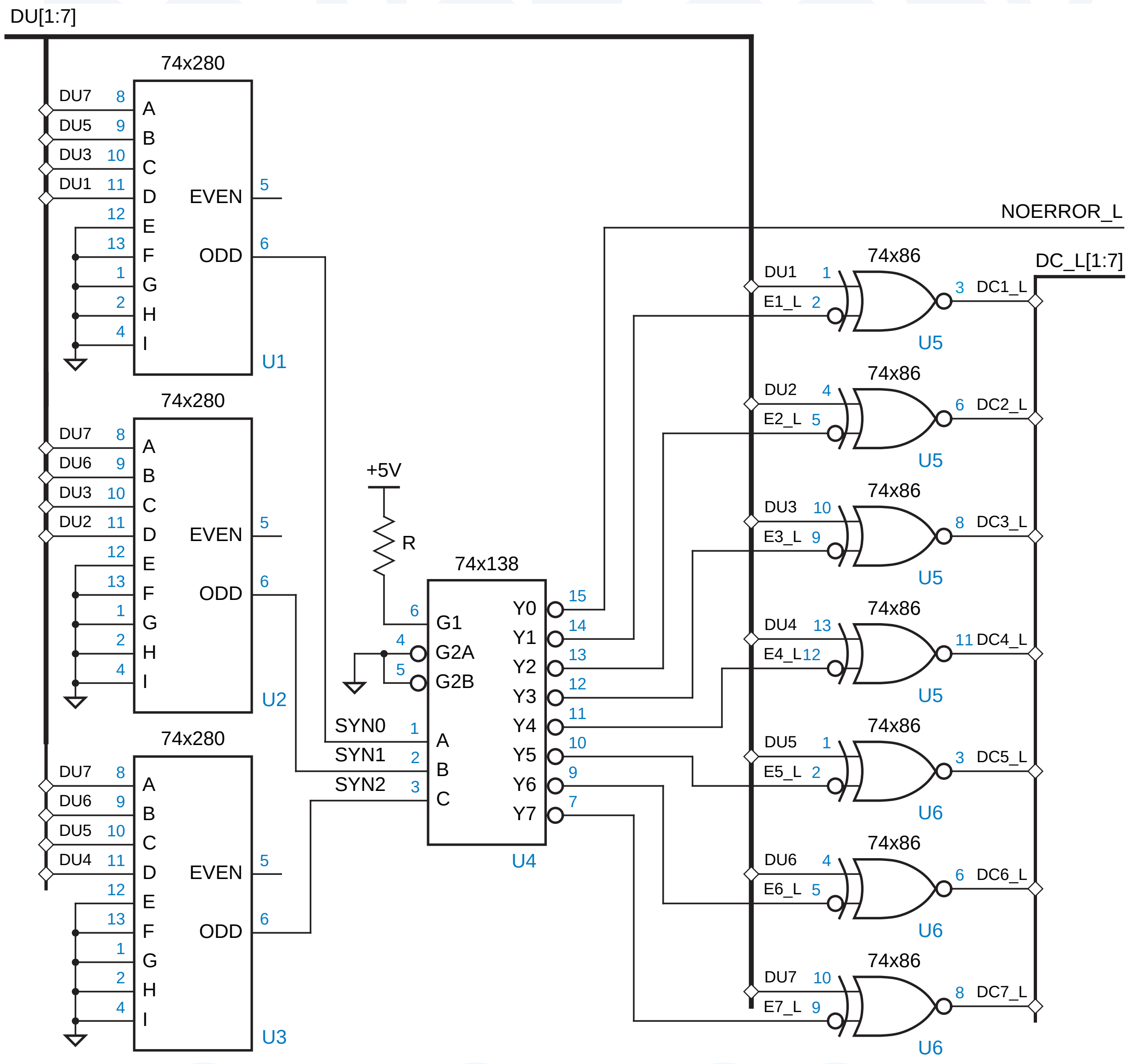


Figure 5-76 Error-correcting circuit for a 7-bit Hamming code.

bit is corrected by complementing it. The corrected code word appears on the DC_L bus.

Note that the active-low outputs of the ’138 led us to use an active-low DC_L bus. If we required an active-high DC bus, we could have put a discrete inverter on each XOR input or output, or used a decoder with active-high outputs, or used XNOR gates.

5.8.5 Exclusive OR Gates and Parity Circuits in ABEL and PLDs

The Exclusive OR function is denoted in ABEL by the $ operator, and its complement, the Exclusive NOR function, is denoted by !$. In principle, these operators may be used freely in ABEL expressions. For example, you could specify a PLD output equivalent to the 74x280's EVEN output using the following ABEL equation:

```
EVEN = !(A $ B $ C $ D $ E $ F $ G $ H $ I);
```

However, most PLDs realize expressions using two-level AND-OR logic and have little if any capability of realizing XOR functions directly. Unfortunately, the Karnaugh map of an n-input XOR function is a checkerboard with 2^{n-1} prime implicants. Thus, the sum-of-products realization of the simple equation above requires 256 product terms, well beyond the capability of any PLD.

As we'll see in Section 10.5.2, some PLDs can realize a two-input XOR function directly in a three-level structure combining two independent sums of products. This structure turns out to be useful in the design of counters. To create larger XOR functions, however, a board-level designer must normally use a specialized parity generator/checker component like the 74x280, and an ASIC designer must combine individual XOR gates in a multilevel parity tree similar to Figure 5-73(b) on page 373.

5.8.6 Exclusive OR Gates and Parity Circuits in VHDL

Like ABEL, VHDL provides primitive operators, xor and xnor, for specifying XOR and XNOR functions (xnor was introduced in VHDL-93). For example, Table 5-45 is a dataflow-style program for a 3-input XOR device that uses the xor primitive. It's also possible to specify XOR or parity functions behaviorally, as Table 5-46 does for a 9-input parity function similar to the 74x280.

Table 5-45 Dataflow-style VHDL program for a 3-input XOR device.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity vxor3 is
    port (
        A, B, C: in STD_LOGIC;
        Y: out STD_LOGIC
    );
end vxor3;

architecture vxor3 of vxor3 is
begin
  Y <= A xor B xor C;
end vxor3;
```

Table 5-46 Behavioral VHDL program for a 9-input parity checker.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity parity9 is
    port (
        I: in STD_LOGIC_VECTOR (1 to 9);
        EVEN, ODD: out STD_LOGIC
    );
end parity9;

architecture parity9p of parity9 is
begin
process (I)
  variable p : STD_LOGIC;
  variable j : INTEGER;
  begin
    p := I(1);
    for j in 2 to 9 loop
      if I(j) = '1' then p := not p; end if;
    end loop;
    ODD <= p;
    EVEN <= not p;
  end process;
end parity9p;
```

When a VHDL program containing large XOR functions is synthesized, the synthesis tool will do the best it can to realize the function in the targeted device technology. There's no magic—if we try to target the VHDL program in Table 5-46 to a 16V8 PLD, it still won't fit!

Typical ASIC and FPGA libraries contain two- and three-input XOR and XNOR functions as primitives. In CMOS ASICs, these primitives are usually realized quite efficiently at the transistor level using transmission gates as shown in Exercises 5.73 and 5.75. Fast and compact XOR trees can be built using these primitives. However, typical VHDL synthesis tools are not be smart enough to create an efficient tree structure from a behavioral program like Table 5-46. Instead, we can use a structural program to get exactly what we want.

For example, Table 5-47 is a structural VHDL program for a 9-input XOR function that is equivalent to the 74x280 of Figure 5-74(a) in structure as well as function. In this example, we've used the previously defined `vxor3` component as the basic building block of the XOR tree. In an ASIC, we would replace the `vxor3` with a 3-input XOR primitive from the ASIC library. Also, if a 3-input XNOR were available, we could eliminate the explicit inversion for `Y3N` and instead use the XNOR for `U5`, using the noninverted `Y3` signal as its last input.

Table 5-47 Structural VHDL program for a 74x280-like parity checker.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x280 is
    port (
        I: in STD_LOGIC_VECTOR (1 to 9);
        EVEN, ODD: out STD_LOGIC
    );
end V74x280;

architecture V74x280s of V74x280 is
component vxor3
 port (A, B, C: in STD_LOGIC; Y: out STD_LOGIC);
end component;
signal Y1, Y2, Y3, Y3N: STD_LOGIC;
begin
  U1: vxor3 port map (I(1), I(2), I(3), Y1);
  U2: vxor3 port map (I(4), I(5), I(6), Y2);
  U3: vxor3 port map (I(7), I(8), I(9), Y3);
  Y3N <= not Y3;
  U4: vxor3 port map (Y1, Y2, Y3, ODD);
  U5: vxor3 port map (Y1, Y2, Y3N, EVEN);
end V74x280s;
```

Our final example is a VHDL version of the Hamming decoder circuit of Figure 5-76. A function `syndrome(DU)` is defined to return the 3-bit syndrome of a 7-bit uncorrected data input vector `DU`. In the "main" process, the corrected data output vector `DC` is initially set equal to `DU`. The `CONV_INTEGER` function, introduced in \secref{VHDLconv}, is used to convert the 3-bit syndrome to an integer. If the syndrome is nonzero, the corresponding bit of `DC` is complemented to correct the assumed 1-bit error. If the syndrome is zero, either no error or an undetectable error has occurred; the output `NOERROR` is set accordingly.

5.9 Comparators

Comparing two binary words for equality is a commonly used operation in computer systems and device interfaces. For example, in Figure 2-7(a) on page 52, we showed a system structure in which devices are enabled by comparing a "device select" word with a predetermined "device ID." A circuit that compares two binary words and indicates whether they are equal is called a *comparator*. Some comparators interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between the words. These devices are often called *magnitude comparators*.

comparator

magnitude comparator

Table 5-48 Behavioral VHDL program for Hamming error correction.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity hamcorr is
  port (
DU: IN STD_LOGIC_VECTOR (1 to 7);
DC: OUT STD_LOGIC_VECTOR (1 to 7);
    NOERROR: OUT STD_LOGIC
);
end hamcorr;

architecture hamcorr of hamcorr is
function syndrome (D: STD_LOGIC_VECTOR)
    return STD_LOGIC_VECTOR is
  variable SYN: STD_LOGIC_VECTOR (2 downto 0);
begin
  SYN(0) := D(1) xor D(3) xor D(5) xor D(7);
  SYN(1) := D(2) xor D(3) xor D(6) xor D(7);
  SYN(2) := D(4) xor D(5) xor D(6) xor D(7);
  return(SYN);
end syndrome;

begin
process (DU)
  variable SYN: STD_LOGIC_VECTOR (2 downto 0);
  variable i: INTEGER;
  begin
    DC <= DU;
    i := CONV_INTEGER(syndrome(DU));
    if i = 0 then NOERROR <= '1';
    else NOERROR <= '0'; DC(i) <= not DU(i); end if;
  end process;
end hamcorr;
```

5.9.1 Comparator Structure

EXCLUSIVE OR and EXCLUSIVE NOR gates may be viewed as 1-bit comparators. Figure 5-77(a) shows an interpretation of the 74x86 XOR gate as a 1-bit comparator. The active-high output, DIFF, is asserted if the inputs are different. The outputs of four XOR gates are ORed to create a 4-bit comparator in (b). The DIFF output is asserted if any of the input-bit pairs are different. Given enough XOR gates and wide enough OR gates, comparators with any number of input bits can be built.

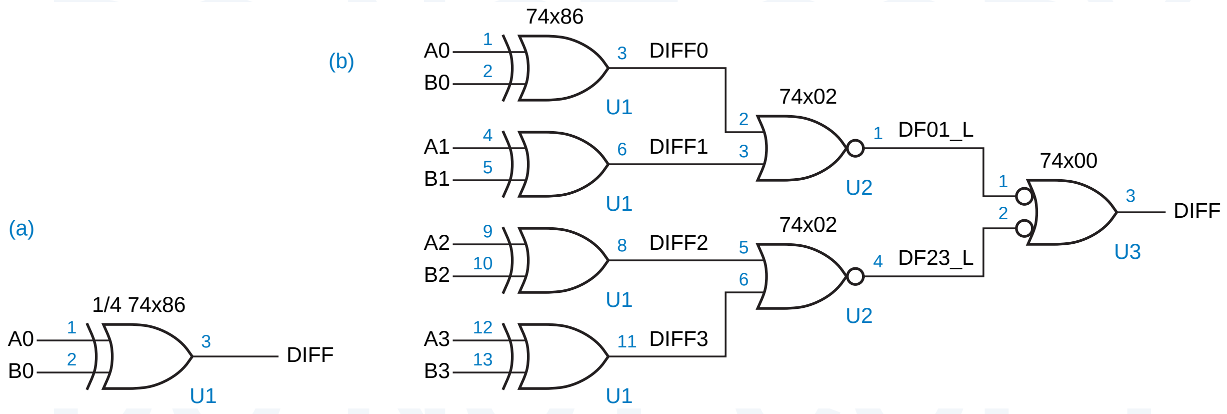


Figure 5-77 Comparators using the 74x86: (a) 1-bit comparator; (b) 4-bit comparator.

AN ITERATIVE COMPARATOR

The *n*-bit comparators in the preceding subsection might be called *parallel comparators* because they look at each pair of input bits simultaneously and deliver the 1-bit comparison results in parallel to an *n*-input OR or AND function. It is also possible to design an “iterative comparator” that looks at its bits one at a time using a small, fixed amount of logic per bit. Before looking at the iterative comparator design, you should understand the general class of “iterative circuits” described in the next subsection.

5.9.2 Iterative Circuits

An *iterative circuit* is a special type of combinational circuit, with the structure shown in Figure 5-78. The circuit contains *n* identical modules, each of which has both *primary inputs and outputs* and *cascading inputs and outputs*. The leftmost cascading inputs are called *boundary inputs* and are connected to fixed logic values in most iterative circuits. The rightmost cascading outputs are called *boundary outputs* and usually provide important information.

iterative circuit

primary inputs and outputs

cascading inputs and outputs

boundary outputs

Iterative circuits are well suited to problems that can be solved by a simple iterative algorithm:

1. Set C_0 to its initial value and set i to 0.
2. Use C_i and PI_i to determine the values of PO_i and C_{i+1}.
3. Increment i.
4. If $i < n$, go to step 2.

In an iterative circuit, the loop of steps 2–4 is “unwound” by providing a separate combinational circuit that performs step 2 for each value of i.

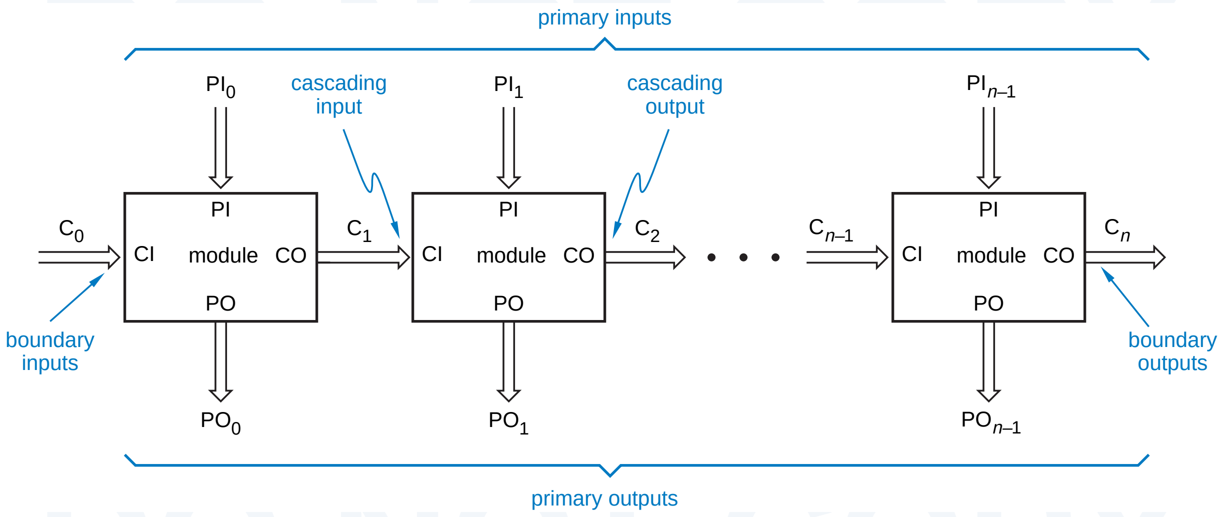


Figure 5-78 General structure of an iterative combinational circuit.

Examples of iterative circuits are the comparator circuit in the next subsection and the ripple adder in Section 5.10.2. The 74x85 4-bit comparator and the 74x283 4-bit adder are examples of MSI circuits that can be used as the individual modules in a larger iterative circuit. In \secref{itvsseq} we'll explore the relationship between iterative circuits and corresponding sequential circuits that execute the 4-step algorithm above in discrete time steps.

5.9.3 An Iterative Comparator Circuit

Two *n*-bit values *X* and *Y* can be compared one bit at a time using a single bit EQ_i at each step to keep track of whether all of the bit-pairs have been equal so far:

1. Set EQ_0 to 1 and set *i* to 0.
2. If EQ_i is 1 and X_i and Y_i are equal, set EQ_{i+1} to 1. Else set EQ_{i+1} to 0.
3. Increment *i*.
4. If $i < n$, go to step 2.

Figure 5-79 shows a corresponding iterative circuit. Note that this circuit has no primary outputs; the boundary output is all that interests us. Other iterative circuits, such as the ripple adder of Section 5.10.2, have primary outputs of interest.

Given a choice between the iterative comparator circuit in this subsection and one of the parallel comparators shown previously, you would probably prefer the parallel comparator. The iterative comparator saves little if any cost, and it's very slow because the cascading signals need time to "ripple" from the leftmost to the rightmost module. Iterative circuits that process more than one bit

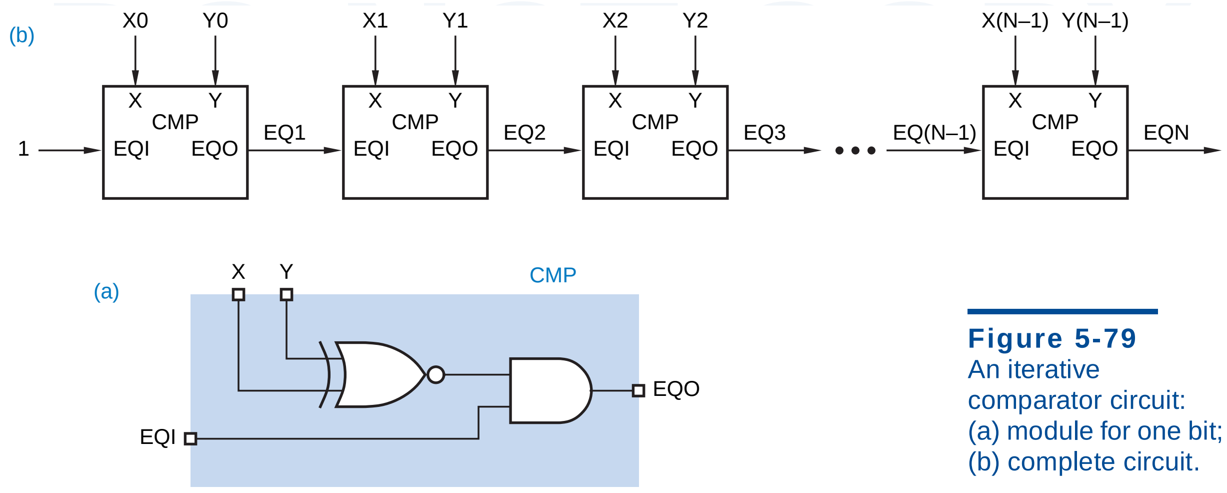


Figure 5-79 An iterative comparator circuit: (a) module for one bit; (b) complete circuit.

at a time, using modules like the 74x85 4-bit comparator and 74x283 4-bit adder, are much more likely to be used in practical designs.

5.9.4 Standard MSI Comparators

Comparator applications are common enough that several MSI comparators have been developed commercially. The *74x85* is a 4-bit comparator with the logic symbol shown in Figure 5-80. It provides a greater-than output (AGTBOUT) and a less-than output (ALTBOUT) as well as an equal output (AEQBOUT). The ’85 also has *cascading inputs* (AGTBIN, ALTBIN, AEQBIN) for combining multiple ’85s to create comparators for more than four bits. Both the cascading inputs and the outputs are arranged in a 1-out-of-3 code, since in normal operation exactly one input and one output should be asserted.

74x85

cascading inputs

The cascading inputs are defined so the outputs of an ’85 that compares less-significant bits are connected to the inputs of an ’85 that compares more-

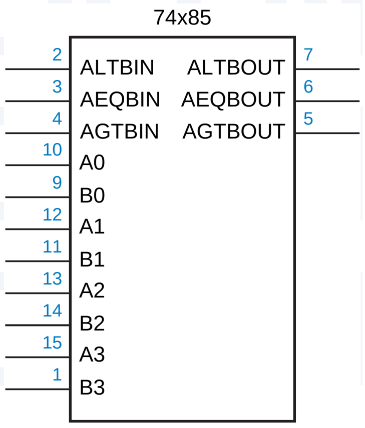


Figure 5-80 Traditional logic symbol for the 74x85 4-bit comparator.

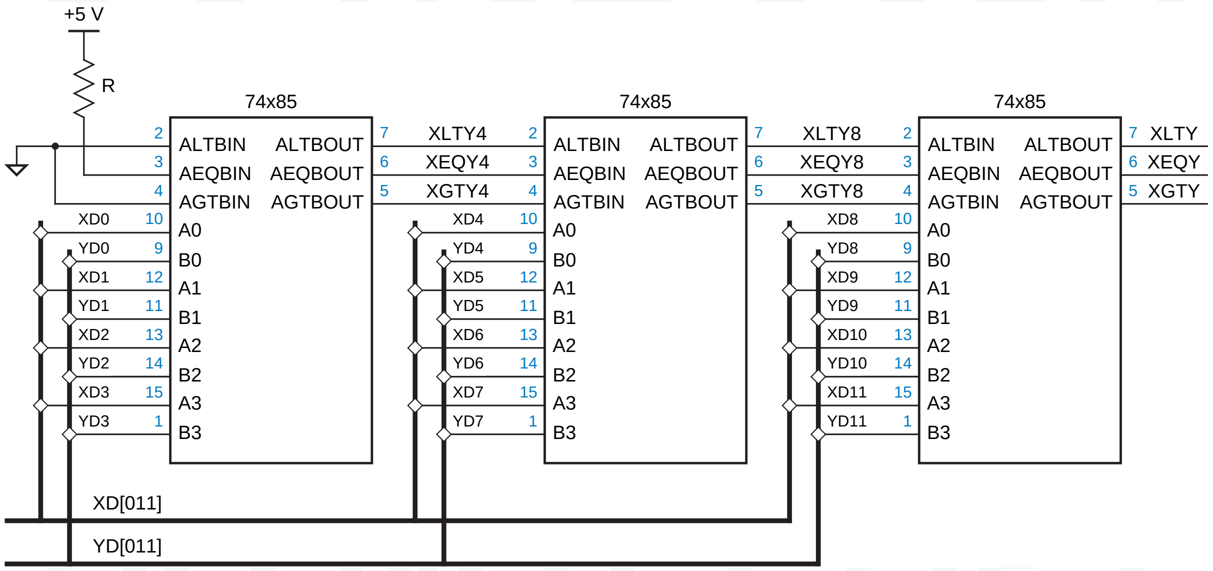


Figure 5-81 A 12-bit comparator using 74x85s.

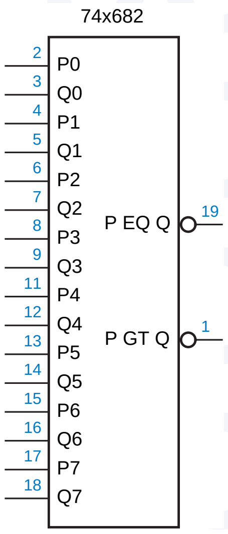


Figure 5-82 Traditional logic symbol for the 74x682 8-bit comparator.

significant bits, as shown in Figure 5-81 for a 12-bit comparator. This is an iterative circuit according to the definition in Section 5.9.2. Each ’85 develops its cascading outputs roughly according to the following pseudo-logic equations:

$$\begin{aligned}\text{AGTBOUT} &= (\text{A} > \text{B}) + (\text{A} = \text{B}) \cdot \text{AGTBIN}\\ \text{AEQBOUT} &= (\text{A} = \text{B}) \cdot \text{AEQBIN}\\ \text{ALTBOUT} &= (\text{A} < \text{B}) + (\text{A} = \text{B}) \cdot \text{ALTBIN}\end{aligned}$$

The parenthesized subexpressions above are not normal logic expressions, but indicate an arithmetic comparison that occurs between the A3–A0 and B3–B0 inputs. In other words, AGTBOUT is asserted if A > B or if A = B and AGTBIN is asserted (if the higher-order bits are equal, we have to look at the lower-order bits for the answer). We’ll see this kind of expression again when we look at ABEL comparator design in Section 5.9.5. The arithmetic comparisons can be expressed using normal logic expressions, for example,

$$\begin{aligned}(\text{A} > \text{B}) = {} & \text{A3} \cdot \text{B3}' + \\ & (\text{A3} \oplus \text{B3})' \cdot \text{A2} \cdot \text{B2}' + \\ & (\text{A3} \oplus \text{B3})' \cdot (\text{A2} \oplus \text{B2})' \cdot \text{A1} \cdot \text{B1}' + \\ & (\text{A3} \oplus \text{B3})' \cdot (\text{A2} \oplus \text{B2})' \cdot (\text{A1} \oplus \text{B1})' \cdot \text{A0} \cdot \text{B0}'\end{aligned}$$

Such expressions must be substituted into the pseudo-logic equations above to obtain genuine logic equations for the comparator outputs.

Several 8-bit MSI comparators are also available. The simplest of these is the *74x682*, whose logic symbol is shown in Figure 5-82 and whose internal

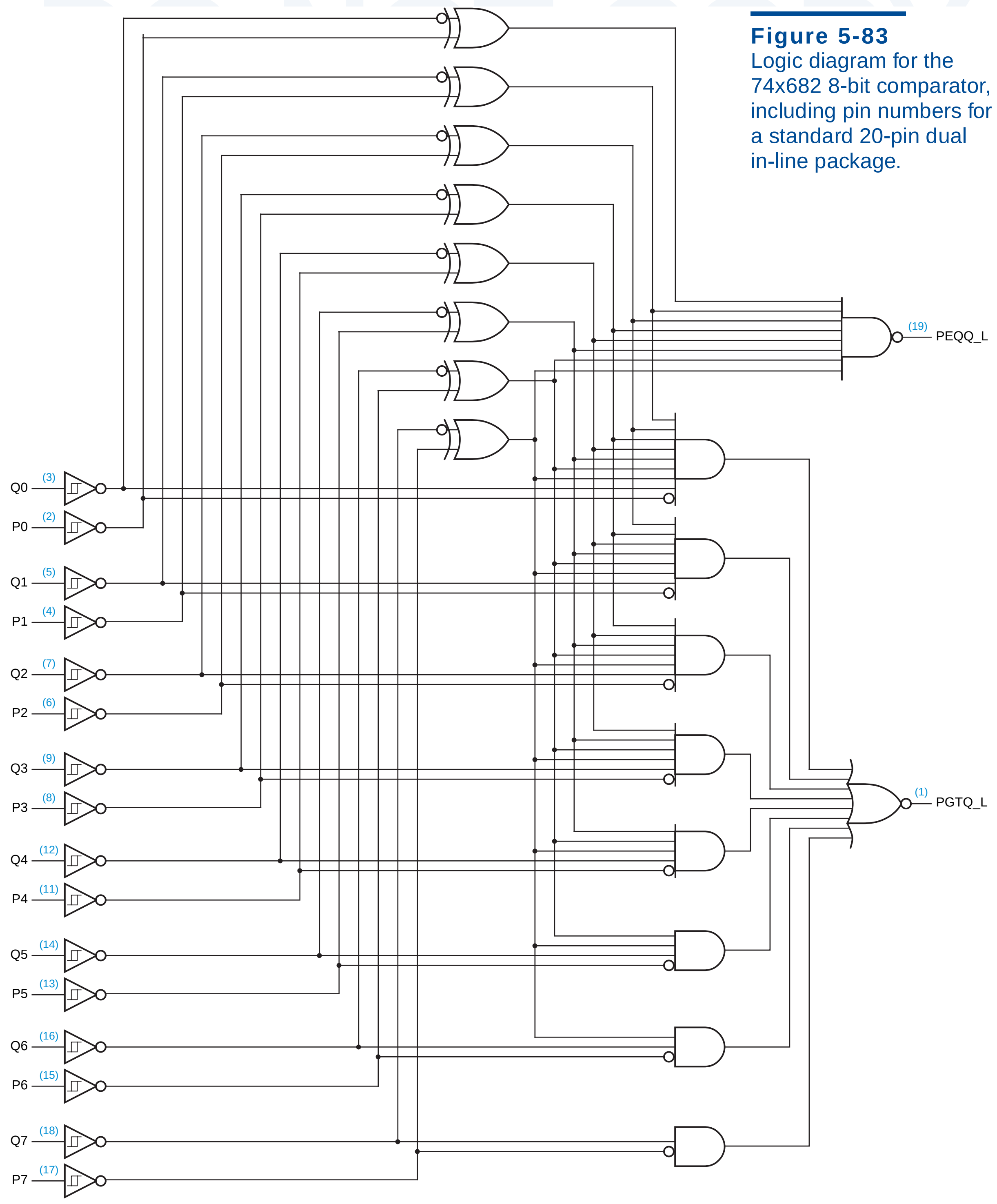
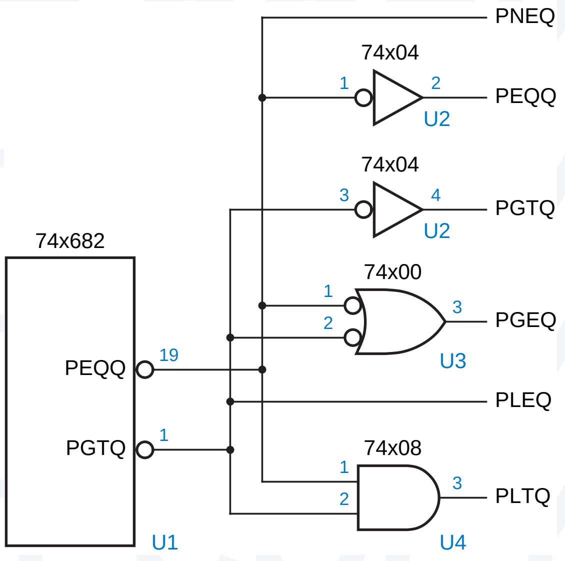


Figure 5-83
Logic diagram for the 74x682 8-bit comparator, including pin numbers for a standard 20-pin dual in-line package.

logic diagram is shown in Figure 5-83. The top half of the circuit checks the two 8-bit input words for equality. Each XNOR-gate output is asserted if its inputs are equal, and the PEQQ_L output is asserted if all eight input-bit pairs are equal. The bottom half of the circuit compares the input words arithmetically, and asserts /PGTQ if P[7–0] > Q[7–0].

Unlike the 74x85, the 74x682 does not have cascading inputs. Also unlike the ’85, the ’682 does not provide a “less than” output. However, any desired condition, including ≤ and ≥, can be formulated as a function of the PEQQ_L and PGTQ_L outputs, as shown in Figure 5-84.

Figure 5-84
Arithmetic conditions derived from 74x682 outputs.



COMPARING COMPARATORS

The individual 1-bit comparators (XNOR gates) in the ’682 are drawn in the opposite sense as the examples of the preceding subsection—outputs are asserted for *equal* inputs and then ANDed, rather than asserted for *different* inputs and then ORed. We can look at a comparator’s function either way, as long as we’re consistent.

5.9.5 Comparators in ABEL and PLDs

Comparing two sets for equality or inequality is very easy to do in ABEL using the “==” or “!=” operator in a relational expression. The only restriction is that the two sets must have an equal number of elements. Thus, given the relational expression “A!=B” where A and B are sets each with *n* elements, the compiler generates the logic expression

```
(A1 $ B1)  # (A2 $ B2)  # ... # (An $ Bn)
```

The logic expression for “A==B”is just the complement of the one above.

In the preceding logic expression, it takes one 2-input XOR function to compare each bit. Since a 2-input XOR function can be realized as a sum of two product terms, the complete expression can be realized in a PLD as a relatively modest sum of 2*n* product terms:

```
(A1&!B1 # !A1&B1) # (A2&!B2 # !A2&&B2) # ... # (An&!Bn # !An&&Bn)
```

Although ABEL has relational operators for less-than and greater-than comparisons, the resulting logic expressions are not so small or easy to derive. For example, consider the relational expression "`A<B`", where `[An..A1]` and `[Bn..B1]` are sets with *n* elements. To construct the corresponding logic expression, ABEL first constructs *n* equations of the form

```
Li = (!Ai & (Bi # Li-1) # (Ai & Bi & Li-1)
```

for *i* = 1 to *n* and `L0` = 0 by definition. This is, in effect, an iterative definition of the less-than function, starting with the least-significant bit. Each `Li` equation says that, as of bit *i*, `A` is less than `B` if `Ai` is 0 and `Bi` is 1 or `A` was less than `B` as of the previous bit, or if `Ai` and `Bi` are both 1 and `A` was less than `B` as of the previous bit.

The logic expression for "`A<B`" is simply the equation for `Ln`. So, after creating the *n* equations above, ABEL collapses them into a single equation for `Ln` involving only elements of `A` and `B`. It does this by substituting the `Ln-1` equation into the right-hand side of the `Ln` equation, then substituting the `Ln-2` equation into this result, and so on, until substituting 0 for `L0`. Finally, it derives a minimal sum-of-products expression from the result.

Collapsing an iterative circuit into a two-level sum-of-products realization usually creates an exponential expansion of product terms. The "<" comparison function follows this pattern, requiring 2^n-1 product terms for an *n*-bit comparator. Thus, comparators larger than a few bits cannot be realized practically in one pass through a PLD.

The results for ">" comparators are identical, of course, and logic expressions for ">="and "<=" are at least as bad, being the complements of the expressions for "<" and ">". If we use a PLD with output polarity control, the inversion is free and the number of product terms is the same; otherwise, the minimal number of product terms after inverting is $2^n+2^{n-1}-1$.

5.9.6 Comparators in VHDL

VHDL has comparison operators for all of its built-in types. *Equality* (`=`) and *inequality* (`/=`) operators apply to all types; for array and record types, the operands must have equal size and structure, and the operands are compared component by component. We have used the equality operator to compare a signal or signal vector with a constant value in many examples in this chapter. If we compare two signals or variables, the synthesis engine generates equations similar to ABEL's in the preceding subsection.

equality, `=`
inequality, `/=`

VHDL's other comparison operators, >, <, >=, and <=, apply only to integer types, enumerated types (such as `STD_LOGIC`), and one-dimensional arrays of enumeration or integer types. Integer order from smallest to largest is the natural ordering, from minus infinity to plus infinity, and enumerated types use the ordering in which the elements of the type were defined, from first to last (unless you explicitly change the enumeration encoding using a command specific to the synthesis engine, in which case the ordering is that of your encoding).

The ordering for array types is defined iteratively, starting with the *leftmost* element in each array. Arrays are always compared from left to right, regardless of the order of their index range ("`to`" or "`downto`"). The order of the leftmost pair of unequal elements is the order of the array. If the arrays have unequal lengths and all the elements of the shorter array match the corresponding elements of the longer one, then the shorter array is considered to be the smaller.

The result of all this is that the built-in comparison operators compare equal-length arrays of type `BIT_VECTOR` or `STD_LOGIC_VECTOR` as if they represented unsigned integers. If the arrays have different lengths, then the operators do *not* yield a valid arithmetic comparison, what you'd get by extending the shorter array with zeroes on the left; more on this in a moment.

Table 5-49 is a VHDL program that produces all of the comparison outputs for comparing two 8-bit unsigned integers. Since the two input vectors `A` and `B` have equal lengths, the program produces the desired results.

Table 5-49 Behavioral VHDL program for comparing 8-bit unsigned integers.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity vcompare is
    port (
        A, B: in STD_LOGIC_VECTOR (7 downto 0);
        EQ, NE, GT, GE, LT, LE: out STD_LOGIC
    );
end vcompare;

architecture vcompare_arch of vcompare is
begin
process (A, B)
  begin
   EQ <= '0'; NE <= '0'; GT <= '0'; GE <= '0'; LT <= '0'; LE <= '0';
    if A = B then EQ <= '1'; end if;
    if A /= B then NE <= '1'; end if;
    if A > B then GT <= '1'; end if;
    if A >= B then GE <= '1'; end if;
    if A < B then LT <= '1'; end if;
    if A <= B then LE <= '1'; end if;
  end process;
end vcompare_arch
```

To allow more flexible comparisons and arithmetic operations, the IEEE has created a standard package, IEEE_std_logic_arith, which defines two important new types and a host of comparison and arithmetic functions that operate on them. The two new types are SIGNED and UNSIGNED:

```
type SIGNED is array (NATURAL range <> of STD_LOGIC;
type UNSIGNED is array (NATURAL range <> of STD_LOGIC;
```

As you can see, both types are defined just indeterminate-length arrays of STD_LOGIC, no different from STD_LOGIC_VECTOR. The important thing is that the package also defines new comparison functions that are invoked when either or both comparison operands have one of the new types. For example, it defines eight new “less-than” functions with the following combinations of parameters:

```
function "<" (L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<" (L: SIGNED; R: SIGNED) return BOOLEAN;
function "<" (L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<" (L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<" (L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<" (L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;
function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;
```

Thus, the “<” operator can be used with any combination of SIGNED, UNSIGNED, and INTEGER operands; the compiler selects the function whose parameter types match the actual operands. Each of the functions is defined in the package to do the “right thing,” including making the appropriate extensions and conversions when operands of different sizes or types are used. Similar functions are provided for the other five relational operators, =, /=, <=, >, and >=.

Using the IEEE_std_logic_arith package, you can write programs like the one in Table 5-50. Its 8-bit input vectors, A, B, C, and D, have three different types. In the comparisons involving A, B, and C, the compiler automatically selects the correct version of the comparison function; for example, for A<B” it selects the first “<” function above, because both operands have type UNSIGNED.

In the comparisons involving D, explicit type conversions are used. The assumption is that the designer wants this particular STD_LOGIC_VECTOR to be interpreted as UNSIGNED in one case and SIGNED in another. The important thing to understand here is that the IEEE_std_logic_arith package does not make any assumptions about how STD_LOGIC_VECTORs are to be interpreted; the user must specify the conversion.

Two other packages, STD_LOGIC_SIGNED and STD_LOGIC_UNSIGNED, do make assumptions and are useful if all STD_LOGIC_VECTORs are to be interpreted the same way. Each package contains three versions of each comparison function so that STD_LOGIC_VECTORs are interpreted as SIGNED or UNSIGNED, respectively, when compared with each other or with integers.

Table 5-50 Behavioral VHDL program for comparing 8-bit integers of various types.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vcompa is
    port (
        A, B: in UNSIGNED (7 downto 0);
        C: in SIGNED (7 downto 0);
        D: in STD_LOGIC_VECTOR (7 downto 0);
        A_LT_B, B_GE_C, A_EQ_C, C_NEG, D_BIG, D_NEG: out STD_LOGIC
    );
end vcompa;

architecture vcompa_arch of vcompa is
begin
process (A, B, C, D)
  begin
    A_LT_B <= '0'; B_GE_C <= '0'; A_EQ_C <= '0'; C_NEG <= '0'; D_BIG <= '0'; D_NEG <= '0';
    if A < B then A_LT_B <= '1'; end if;
    if B >= C then B_GE_C <= '1'; end if;
    if A = C then A_EQ_C <= '1'; end if;
    if C < 0 then C_NEG <= '1'; end if;
    if UNSIGNED(D) > 200 then D_BIG <= '1'; end if;
    if SIGNED(D) < 0 then D_NEG <= '1'; end if;
  end process;
end vcompa_arch;
```

When a comparison function is specified in VHDL, it takes just as many product terms as in ABEL to realize the function as a two-level sum of products. However, most VHDL synthesis engines will realize the comparator as an iterative circuit with far fewer gates, albeit more levels of logic. Also, better synthesis engines can detect opportunities to eliminate entire comparator circuits. For example, in the program of Table 5-49 on page 388, the NE, GE, and LE outputs could be realized with one inverter each, as the complements of the EQ, LT, and GT outputs, respectively.

*5.10 Adders, Subtractors, and ALUs

adder

subtractor

Addition is the most commonly performed arithmetic operation in digital systems. An *adder* combines two arithmetic operands using the addition rules described in Chapter 2. As we showed in Section 2.6, the same addition rules and therefore the same adders are used for both unsigned and two's-complement numbers. An adder can perform subtraction as the addition of the minuend and the complemented (negated) subtrahend, but you can also build *subtractor* circuits that perform subtraction directly. MSI devices called ALUs, described in Section 5.10.6, perform addition, subtraction, or any of several other operations according to an operation code supplied to the device.

*5.10.1 Half Adders and Full Adders

half adder

The simplest adder, called a *half adder*, adds two 1-bit operands X and Y, producing a 2-bit sum. The sum can range from 0 to 2, which requires two bits to express. The low-order bit of the sum may be named HS (half sum), and the high-order bit may be named CO (carry out). We can write the following equations for HS and CO:

$$\begin{aligned} \mathrm{HS} &= \mathrm{X} \oplus \mathrm{Y} \\ &= \mathrm{X} \cdot \mathrm{Y}' + \mathrm{X}' \cdot \mathrm{Y} \\ \mathrm{CO} &= \mathrm{X} \cdot \mathrm{Y} \end{aligned}$$

full adder

To add operands with more than one bit, we must provide for carries between bit positions. The building block for this operation is called a *full adder*. Besides the addend-bit inputs X and Y, a full adder has a carry-bit input, CIN. The sum of the three inputs can range from 0 to 3, which can still be expressed with just two output bits, S and COUT, having the following equations:

$$\begin{aligned} \mathrm{S} &= \mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{CIN} \\ &= \mathrm{X} \cdot \mathrm{Y}' \cdot \mathrm{CIN}' + \mathrm{X}' \cdot \mathrm{Y} \cdot \mathrm{CIN}' + \mathrm{X}' \cdot \mathrm{Y}' \cdot \mathrm{CIN} + \mathrm{X} \cdot \mathrm{Y} \cdot \mathrm{CIN} \\ \mathrm{COUT} &= \mathrm{X} \cdot \mathrm{Y} + \mathrm{X} \cdot \mathrm{CIN} + \mathrm{Y} \cdot \mathrm{CIN} \end{aligned}$$

Here, S is 1 if an odd number of the inputs are 1, and COUT is 1 if two or more of the inputs are 1. These equations represent the same operation that was specified by the binary addition table in Table 2-3 on page 28.

One possible circuit that performs the full-adder equations is shown in Figure 5-85(a). The corresponding logic symbol is shown in (b). Sometimes the symbol is drawn as shown in (c), so that cascaded full adders can be drawn more neatly, as in the next subsection.

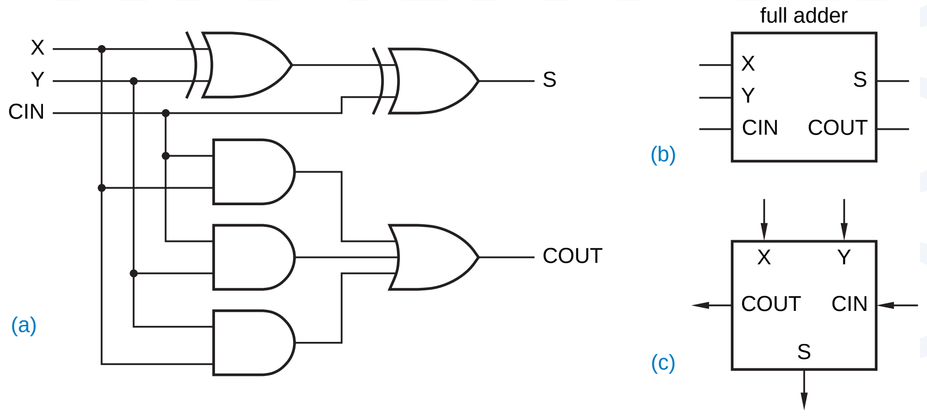


Figure 5-85 Full adder: (a) gate-level circuit diagram; (b) logic symbol; (c) alternate logic symbol suitable for cascading.

*5.10.2 Ripple Adders

ripple adder

Two binary words, each with *n* bits, can be added using a *ripple adder*—a cascade of *n* full-adder stages, each of which handles one bit. Figure 5-86 shows the circuit for a 4-bit ripple adder. The carry input to the least significant bit (c_0) is normally set to 0, and the carry output of each full adder is connected to the carry input of the next most significant full adder. The ripple adder is a classic example of an iterative circuit as defined in Section 5.9.2.

A ripple adder is slow, since in the worst case a carry must propagate from the least significant full adder to the most significant one. This occurs if, for example, one addend is 11 … 11 and the other is 00 … 01. Assuming that all of the addend bits are presented simultaneously, the total worst-case delay is

$$t_{\mathrm{ADD}} = t_{\mathrm{XYCout}} + (n-2) \cdot t_{\mathrm{CinCout}} + t_{\mathrm{CinS}}$$

where t_{XYCout} is the delay from X or Y to COUT in the least significant stage, t_{CinCout} is the delay from CIN to COUT in the middle stages, and t_{CinS} is the delay from CIN to S in the most significant stage.

A faster adder can be built by obtaining each sum output s_i with just two levels of logic. This can be accomplished by writing an equation for s_i in terms of x_0–x_i, y_0–y_i, and c_0, “multiplying out” or “adding out” to obtain a sum-of-products or product-of-sums expression, and building the corresponding AND-OR or OR-AND circuit. Unfortunately, beyond s_2, the resulting expressions have too many terms, requiring too many first-level gates and more inputs than typically possible on the second-level gate. For example, even assuming that $c_0 = 0$, a two-level AND-OR circuit for s_2 requires fourteen 4-input ANDs, four 5-input ANDs, and an 18-input OR gate; higher-order sum bits are even worse. Nevertheless, it is possible to build adders with just a few levels of delay using a more reasonable number of gates, as we’ll see in Section 5.10.4.

*5.10.3 Subtractors

full subtractor

A binary subtraction operation analogous to binary addition was also specified in Table 2-3 on page 28. A *full subtractor* handles one bit of the binary subtraction algorithm, having input bits X (minuend), Y (subtrahend), and BIN (borrow

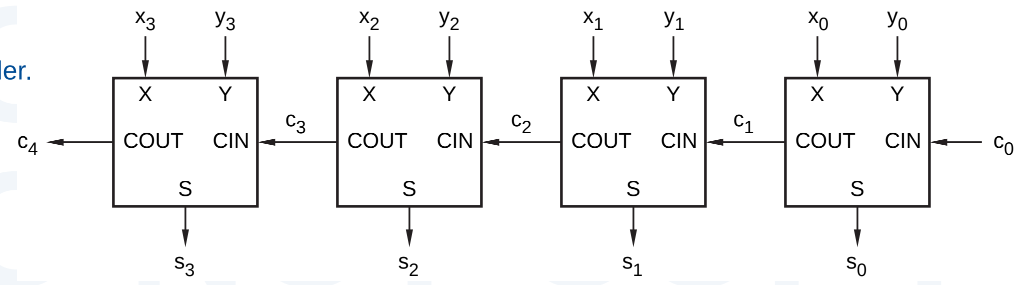


Figure 5-86 A 4-bit ripple adder.

in), and output bits D (difference) and BOUT (borrow out). We can write logic equations corresponding to the binary subtraction table as follows:

$$\begin{aligned} \mathrm{D} &= \mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{BIN} \\ \mathrm{BOUT} &= \mathrm{X}' \cdot \mathrm{Y} + \mathrm{X}' \cdot \mathrm{BIN} + \mathrm{Y} \cdot \mathrm{BIN} \end{aligned}$$

These equations are very similar to equations for a full adder, which should not be surprising. We showed in Section 2.6 that a two's-complement subtraction operation, $X - Y$, can be performed by an addition operation, namely, by adding the two's complement of Y to X. The two's complement of Y is $\overline{Y} + 1$, where $\overline{Y}$ is the bit-by-bit complement of Y. We also showed in Exercise 2.26 that a binary adder can be used to perform an unsigned subtraction operation $X - Y$, by performing the operation $X + \overline{Y} + 1$. We can now confirm that these statements are true by manipulating the logic equations above:

$$\begin{aligned} \mathrm{BOUT} &= \mathrm{X}' \cdot \mathrm{Y} + \mathrm{X}' \cdot \mathrm{BIN} + \mathrm{Y} \cdot \mathrm{BIN} \\ \mathrm{BOUT}' &= (\mathrm{X} + \mathrm{Y}') \cdot (\mathrm{X} + \mathrm{BIN}') \cdot (\mathrm{Y}' + \mathrm{BIN}') && \text{(generalized DeMorgan's theorem)} \\ &= \mathrm{X} \cdot \mathrm{Y}' + \mathrm{X} \cdot \mathrm{BIN}' + \mathrm{Y}' \cdot \mathrm{BIN}' && \text{(multiply out)} \\ \mathrm{D} &= \mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{BIN} \\ &= \mathrm{X} \oplus \mathrm{Y}' \oplus \mathrm{BIN}' && \text{(complementing XOR inputs)} \end{aligned}$$

For the last manipulation, recall that we can complement the two inputs of an XOR gate without changing the function performed.

Comparing with the equations for a full adder, the above equations tell us that we can build a full subtractor from a full adder as shown in Figure 5-87. Just to keep things straight, we've given the full adder circuit in (a) a fictitious name, the "74x999." As shown in (c), we can interpret the function of this same physical circuit to be a full subtractor by giving it a new symbol with active-low borrow in, borrow out, and subtrahend signals.

Thus, to build a ripple subtractor for two *n*-bit active-high operands, we can use *n* 74x999s and inverters, as shown in (d). Note that for the subtraction operation, the borrow input of the least significant bit should be negated (no borrow), which for an active-low input means that the physical pin must be 1 or HIGH. This is just the opposite as in addition, where the same input pin is an active-high carry-in that is 0 or LOW.

By going back to the math in Chapter 2, we can show that this sort of manipulation works for all adder and subtractor circuits, not just ripple adders and subtractors. That is, any *n*-bit adder circuit can be made to function as a subtractor by complementing the subtrahend and treating the carry-in and carry-out signals as borrows with the opposite active level. The rest of this section discusses addition circuits only, with the understanding that they can easily be made to perform subtraction.

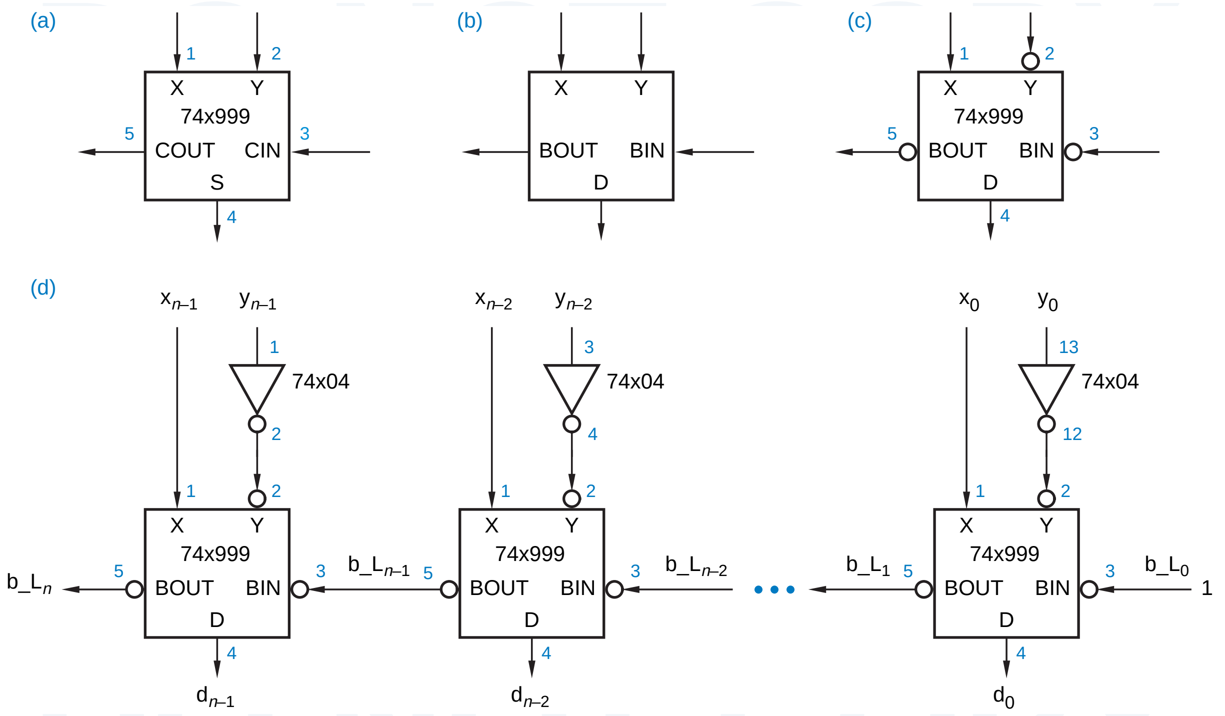


Figure 5-87 Designing subtractors using adders: (a) full adder; (b) full subtractor; (c) interpreting the device in (a) as a full subtractor; (d) ripple subtractor.

*5.10.4 Carry Lookahead Adders

The logic equation for sum bit *i* of a binary adder can actually be written quite simply:

$$s_i = x_i \oplus y_i \oplus c_i$$

More complexity is introduced when we expand c_i above in terms of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0, and we get a real mess expanding the XORs. However, if we're willing to forego the XOR expansion, we can at least streamline the design of c_i logic using ideas of *carry lookahead* discussed in this subsection.

carry lookahead

Figure 5-88 shows the basic idea. The block labeled "Carry Lookahead Logic" calculates c_i in a fixed, small number of logic levels for any reasonable value of *i*. Two definitions are the key to carry lookahead logic:

carry generate

- For a particular combination of inputs x_i and y_i, adder stage *i* is said to *generate* a carry if it produces a carry-out of 1 ($c_{i+1} = 1$) independent of the inputs on $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0.

carry propagate

- For a particular combination of inputs x_i and y_i, adder stage *i* is said to *propagate* carries if it produces a carry-out of 1 ($c_{i+1} = 1$) in the presence of an input combination of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0 that causes a carry-in of 1 ($c_i = 1$).

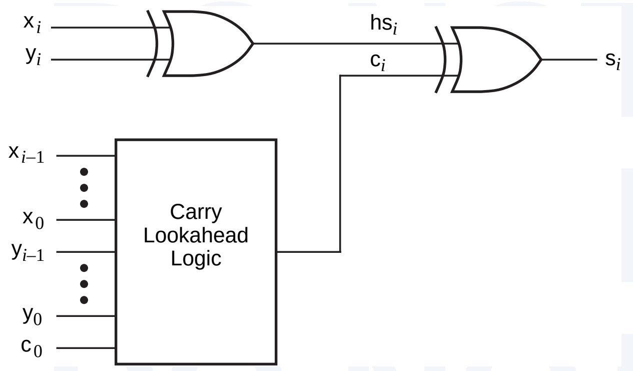


Figure 5-88
Structure of one stage of a carry lookahead adder.

Corresponding to these definitions, we can write logic equations for a carry-generate signal, g_i, and a carry-propagate signal, p_i, for each stage of a carry lookahead adder:

$$\begin{aligned} g_i &= x_i \cdot y_i \\ p_i &= x_i + y_i \end{aligned}$$

That is, a stage unconditionally generates a carry if both of its addend bits are 1, and it propagates carries if at least one of its addend bits is 1. The carry output of a stage can now be written in terms of the generate and propagate signals:

$$c_{i+1} = g_i + p_i \cdot c_i$$

To eliminate carry ripple, we recursively expand the c_i term for each stage, and multiply out to obtain a 2-level AND-OR expression. Using this technique, we can obtain the following carry equations for the first four adder stages:

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
&= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}$$

Each equation corresponds to a circuit with just three levels of delay—one for the generate and propagate signals, and two for the sum-of-products shown. A *carry lookahead adder* uses three-level equations such as these in each adder stage for the block labeled "carry lookahead" in Figure 5-88. The sum output for

carry lookahead adder

a stage is produced by combining the carry bit with the two addend bits for the stage as we showed in the figure. In the next subsection, we'll study some commercial MSI adders and ALUs that use carry lookahead.

*5.10.5 MSI Adders

74x283

74x83

The *74x283* is a 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic, using the carry lookahead technique. Figure 5-89 is a logic symbol for the 74x283. The older *74x83* is identical except for its pinout, which has nonstandard locations for power and ground.

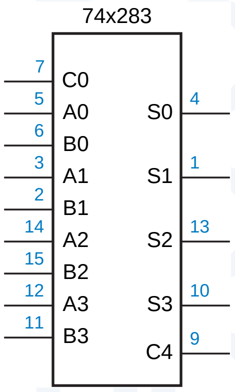


Figure 5-89 Traditional logic symbol for the 74x283 4-bit binary adder.

The logic diagram for the '283, shown in Figure 5-90, has just a few differences from the general carry-lookahead design that we described in the preceding subsection. First of all, its addends are named A and B instead of X and Y; no big deal. Second, it produces active-low versions of the carry-generate ($g_i{}'$) and carry-propagate ($p_i{}'$) signals, since inverting gates are generally faster than noninverting ones. Third, it takes advantage of the fact that we can algebraically manipulate the half-sum equation as follows:

$$\begin{aligned} hs_i &= x_i \oplus y_i \\ &= x_i \cdot y_i' + x_i' \cdot y_i \\ &= x_i \cdot y_i' + x_i \cdot x_i' + x_i' \cdot y_i + y_i \cdot y_i' \\ &= (x_i + y_i) \cdot (x_i' + y_i') \\ &= (x_i + y_i) \cdot (x_i \cdot y_i)' \\ &= p_i \cdot g_i' \end{aligned}$$

Thus, an AND gate with an inverted input can be used instead of an XOR gate to create each half-sum bit.

Finally, the '283 creates the carry signals using an INVERT-OR-AND structure (the DeMorgan equivalent of an AND-OR-INVERT), which has about the same delay as a single CMOS or TTL inverting gate. This requires some explaining, since the carry equations that we derived in the preceding subsection are used in a slightly modified form. In particular, the c_{i+1} equation uses the term $p_i \cdot g_i$ instead of g_i. This has no effect on the output, since p_i is always 1 when g_i is 1. However, it allows the equation to be factored as follows:

$$\begin{aligned} c_{i+1} &= p_i \cdot g_i + p_i \cdot c_i \\ &= p_i \cdot (g_i + c_i) \end{aligned}$$

This leads to the following carry equations, which are used by the circuit :

$$\begin{aligned} c_1 &= p_0 \cdot (g_0 + c_0) \\ c_2 &= p_1 \cdot (g_1 + c_1) \\ &= p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0)) \\ &= p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0) \end{aligned}$$

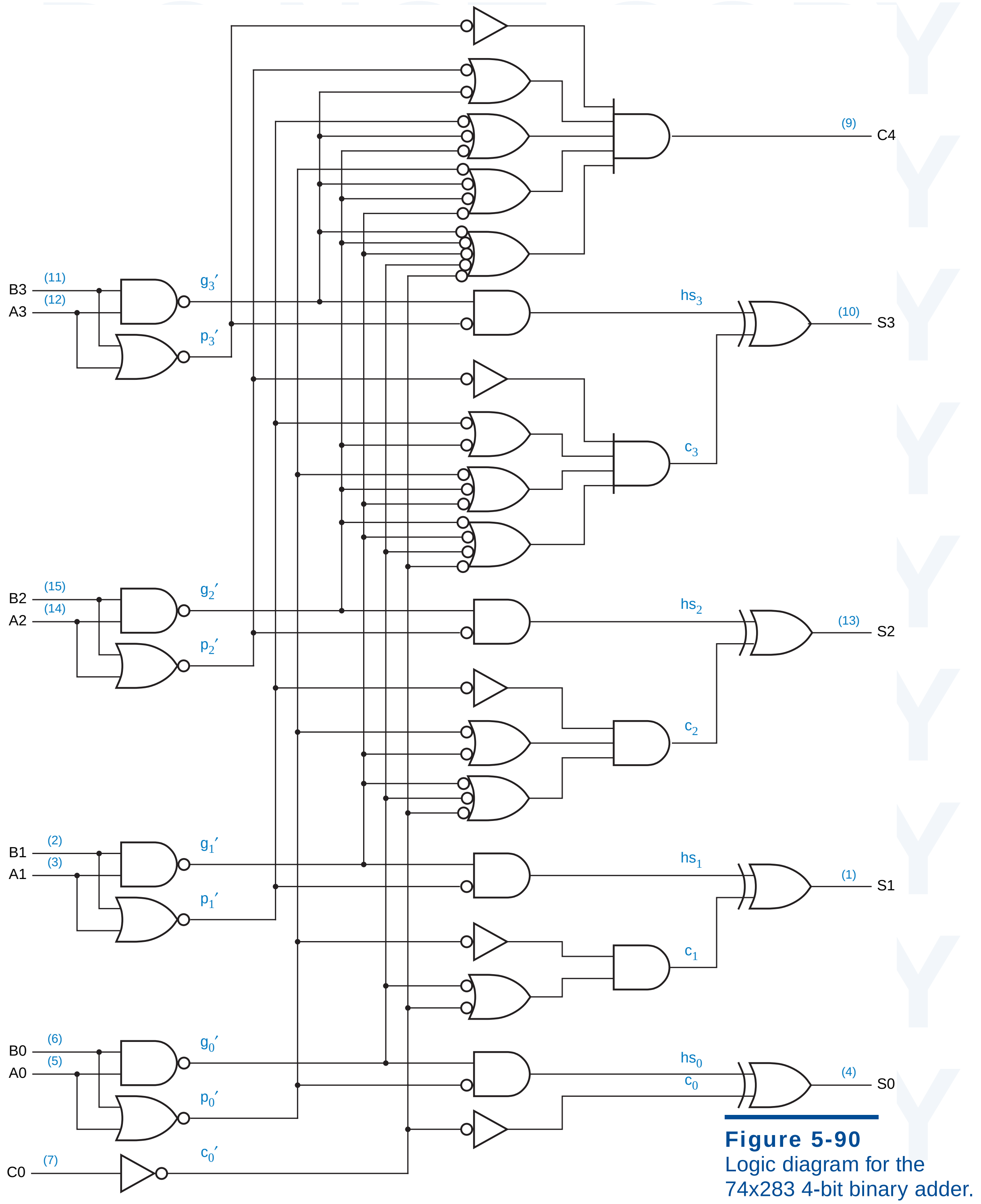


Figure 5-90
Logic diagram for the 74x283 4-bit binary adder.

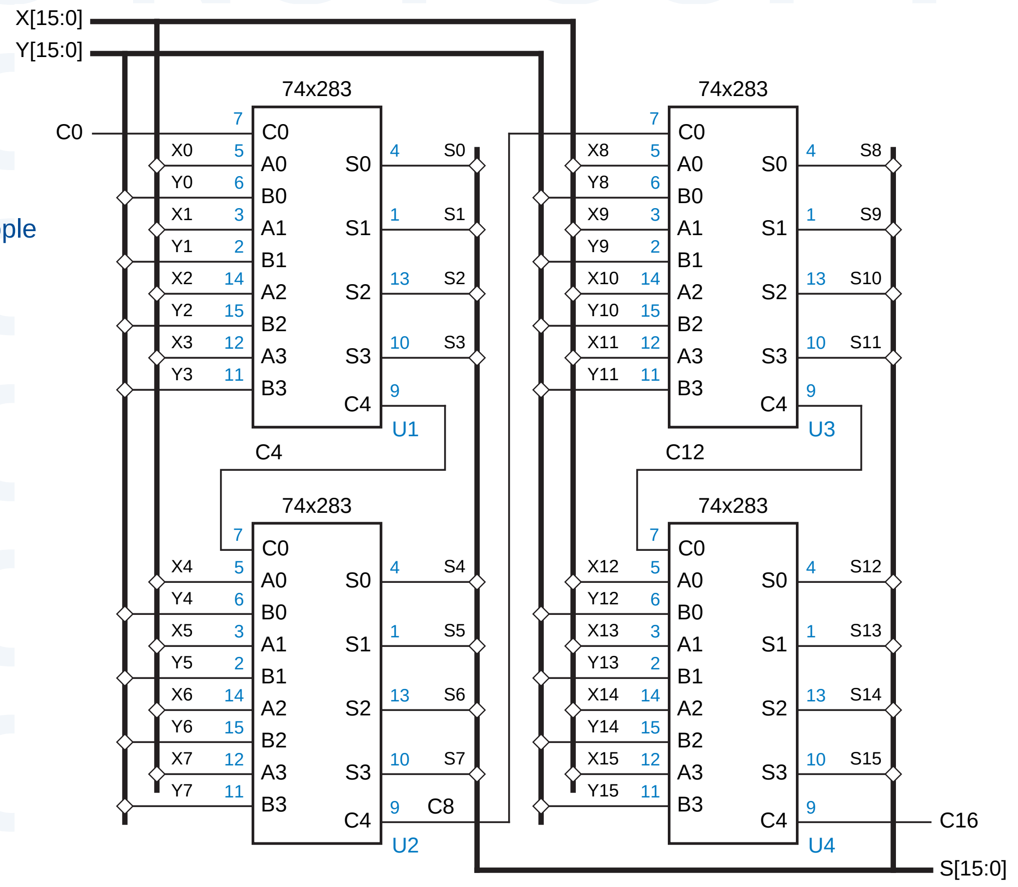
$$\begin{aligned}
c_3 &= p_2 \cdot (g_2 + c_2) \\
&= p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)) \\
&= p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0) \\
c_4 &= p_3 \cdot (g_3 + c_3) \\
&= p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)) \\
&= p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \cdot (g_3 + g_2 + g_1 + g_0 + c_0)
\end{aligned}$$

If you've followed the derivation of these equations and can obtain the same ones by reading the '283 logic diagram, then congratulations, you're up to speed on switching algebra! If not, you may want to review Sections 4.1 and 4.2.

group-ripple adder

The propagation delay from the C0 input to the C4 output of the '283 is very short, about the same as two inverting gates. As a result, fairly fast *group-ripple adders* with more than four bits can be made simply by cascading the carry outputs and inputs of '283s, as shown in Figure 5-91 for a 16-bit adder. The total propagation delay from C0 to C16 in this circuit is about the same as that of eight inverting gates.

Figure 5-91
A 16-bit group-ripple adder.



*5.10.6 MSI Arithmetic and Logic Units

An *arithmetic and logic unit (ALU)* is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of *b*-bit operands. The operation to be performed is specified by a set of function-select inputs. Typical MSI ALUs have 4-bit operands and three to five function select inputs, allowing up to 32 different functions to be performed.

arithmetic and logic unit (ALU)

Figure 5-92 is a logic symbol for the *74x181* 4-bit ALU. The operation performed by the ’181 is selected by the M and S3–S0 inputs, as detailed in Table 5-51. Note that the identifiers A, B, and F in the table refer to the 4-bit words A3–A0, B3–B0, and F3–F0; and the symbols · and + refer to logical AND and OR operations.

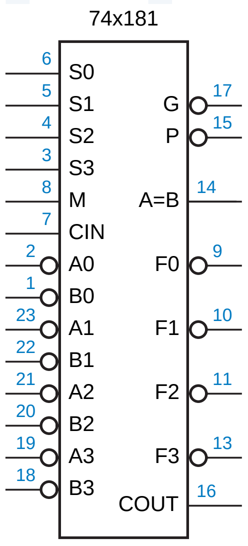
74x181

The 181’s M input selects between arithmetic and logical operations. When M = 1, logical operations are selected, and each output Fi is a function only of the corresponding data inputs, Ai and Bi. No carries propagate between stages, and the CIN input is ignored. The S3–S0 inputs select a particular logical operation; any of the 16 different combinational logic functions on two variables may be selected.

Table 5-51 Functions performed by the 74x181 4-bit ALU.

Inputs				***Function***	
S3	S2	S1	S0	***M = 0 (arithmetic)***	***M = 1 (logic)***
0	0	0	0	F = A minus 1 plus CIN	F = A′
0	0	0	1	F = A · B minus 1 plus CIN	F = A′ + B′
0	0	1	0	F = A · B′ minus 1 plus CIN	F = A′ + B
0	0	1	1	F = 1111 plus CIN	F = 1111
0	1	0	0	F = A plus (A + B′) plus CIN	F = A′ · B′
0	1	0	1	F = A · B plus (A + B′) plus CIN	F = B′
0	1	1	0	F = A minus B minus 1 plus CIN	F = A ⊕ B′
0	1	1	1	F = A + B′ plus CIN	F = A + B′
1	0	0	0	F = A plus (A + B) plus CIN	F = A′ · B
1	0	0	1	F = A plus B plus CIN	F = A ⊕ B
1	0	1	0	F = A · B′ plus (A + B) plus CIN	F = B
1	0	1	1	F = A + B plus CIN	F = A + B
1	1	0	0	F = A plus A plus CIN	F = 0000
1	1	0	1	F = A · B plus A plus CIN	F = A · B′
1	1	1	0	F = A · B′ plus A plus CIN	F = A · B
1	1	1	1	F = A plus CIN	F = A

Figure 5-92 Logic symbol for the 74x181 4-bit ALU.



When M = 0, arithmetic operations are selected, carries propagate between the stages, and CIN is used as a carry input to the least significant stage. For operations larger than four bits, multiple ’181 ALUs may be cascaded like the group-ripple adder in the Figure 5-91, with the carry-out (COUT) of each ALU connected to the carry-in (CIN) of the next most significant stage. The same function-select signals (M, S3–S0) are applied to all the ’181s in the cascade.

To perform two’s-complement addition, we use S3–S0 to select the operation “A plus B plus CIN.” The CIN input of the least-significant ALU is normally set to 0 during addition operations. To perform two’s-complement subtraction, we use S3–S0 to select the operation A minus B minus plus CIN. In this case, the CIN input of the least significant ALU is normally set to 1, since CIN acts as the complement of the borrow during subtraction.

The ’181 provides other arithmetic operations, such as “A minus 1 plus CIN,” that are useful in some applications (e.g., decrement by 1). It also provides a bunch of weird arithmetic operations, such as “A · B′ plus (A + B) plus CIN,” that are almost never used in practice, but that “fall out” of the circuit for free.

Notice that the operand inputs A3_L–A0_L and B3_L–B0_L and the function outputs F3_L–F0_L of the ’181 are active low. The ’181 can also be used with active-high operand inputs and function outputs. In this case, a different version of the function table must be constructed. When M = 1, logical operations are still performed, but for a given input combination on S3–S0, the function obtained is precisely the dual of the one listed in Table 5-51. When M = 0, arithmetic operations are performed, but the function table is once again different. Refer to a ’181 data sheet for more details.

74x381
74x382

Two other MSI ALUs, the *74x381* and *74x382* shown in Figure 5-93, encode their select inputs more compactly, and provide only eight different but useful functions, as detailed in Table 5-52. The only difference between the ’381 and ’382 is that one provides group-carry lookahead outputs (which we explain next), while the other provides ripple carry and overflow outputs.

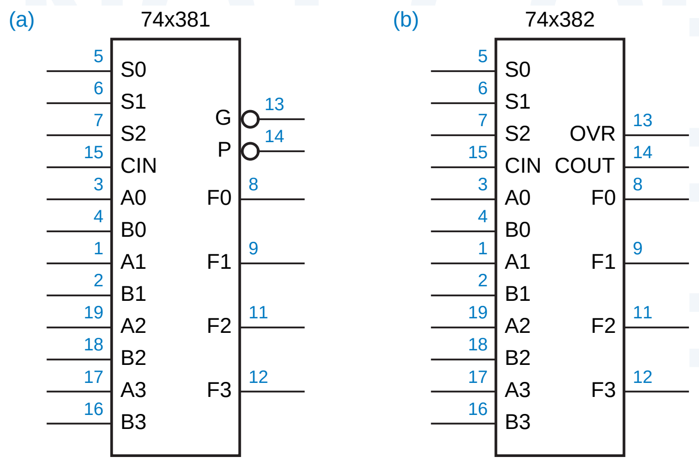


Figure 5-93
Logic symbols for 4-bit ALUs: (a) 74x381; (b) 74x382.

Inputs			
S2	S1	S0	***Function***
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111

Table 5-52 Functions performed by the 74x381 and 74x382 4-bit ALUs.

*5.10.7 Group-Carry Lookahead

group-carry lookahead

The ’181 and ’381 provide *group-carry lookahead* outputs that allow multiple ALUs to be cascaded without rippling carries between 4-bit groups. Like the 74x283, the ALUs use carry lookahead to produce carries internally. However, they also provide G_L and P_L outputs that are carry lookahead signals for the entire 4-bit group. The G_L output is asserted if the ALU generates a carry, that is, if it will produce a carry-out (COUT = 1) whether or not there is a carry-in (CIN = 1):

$$\text{G\_L} = (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0)'$$

The P_L output is asserted if the ALU propagates a carry, that is, if it will produce a carry-out if there is a carry-in:

$$\text{P\_L} = (p_3 \cdot p_2 \cdot p_1 \cdot p_0)'$$

lookahead carry circuit
74x182

When ALUs are cascaded, the group-carry lookahead outputs may be combined in just two levels of logic to produce the carry input to each ALU. A *lookahead carry circuit*, the *74x182* shown in Figure 5-94, performs this operation. The ’182 inputs are C0, the carry input to the least significant ALU (“ALU 0”), and G0–G3 and P0–P3, the generate and propagate outputs of ALUs 0–3. Using these inputs, the ’182 produces carry inputs C1–C3 for ALUs 1–3. Figure 5-95 shows the connections for a 16-bit ALU using four ’381s and a ’182.

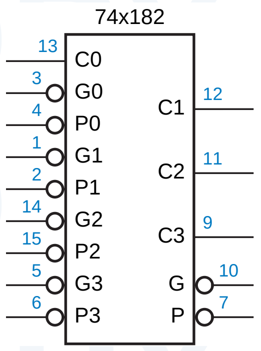


Figure 5-94 Logic symbol for the 74x182 lookahead carry circuit.

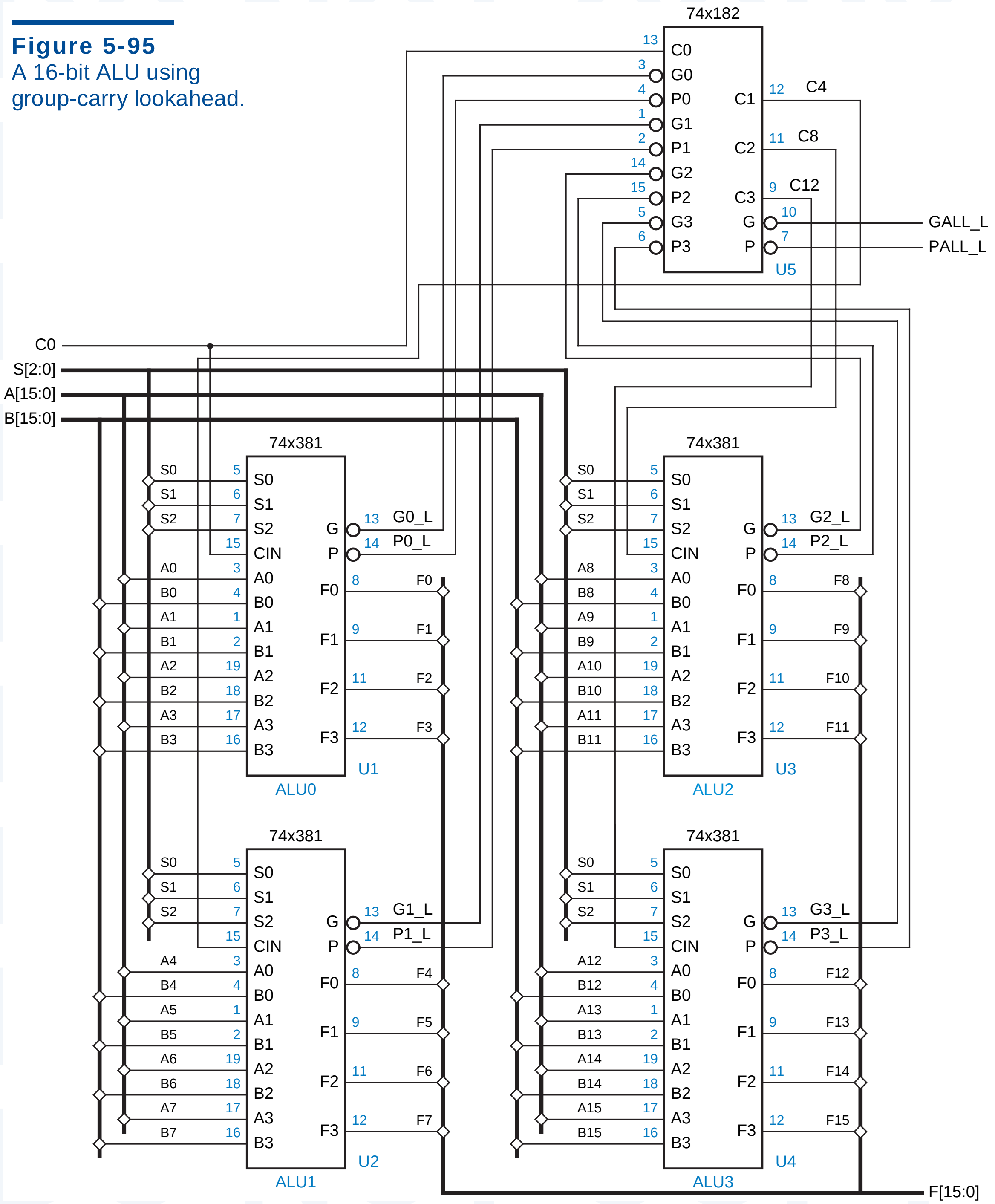
The 182’s carry equations are obtained by “adding out” the basic carry lookahead equation of Section 5.10.4:

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ &= (g_i + p_i) \cdot (g_i + c_i) \end{aligned}$$

Expanding for the first three values of *i*, we obtain the following equations:

$$\begin{aligned} C1 &= (G0+P0) \cdot (G0+C0) \\ C2 &= (G1+P1) \cdot (G1+G0+P0) \cdot (G1+G0+C0) \\ C3 &= (G2+P2) \cdot (G2+G1+P1) \cdot (G2+G1+G0+P0) \cdot (G2+G1+G0+C0) \end{aligned}$$

Figure 5-95
A 16-bit ALU using group-carry lookahead.



The ’182 realizes each of these equations with just one level of delay—an INVERT-OR-AND gate.

When more than four ALUs are cascaded, they may be partitioned into “supergroups,” each with its own ’182. For example, a 64-bit adder would have four supergroups, each containing four ALUs and a ’182. The G_L and P_L outputs of each ’182 can be combined in a next-level ’182, since they indicate whether the supergroup generates or propagates carries:

$$\mathrm{G\_L} = ((\mathrm{G3}+\mathrm{P3}) \cdot (\mathrm{G3}+\mathrm{G2}+\mathrm{P2}) \cdot (\mathrm{G3}+\mathrm{G2}+\mathrm{G1}+\mathrm{P1}) \cdot (\mathrm{G3}+\mathrm{G2}+\mathrm{G1}+\mathrm{G0}))'$$

$$\mathrm{P\_L} = (\mathrm{P0} \cdot \mathrm{P1} \cdot \mathrm{P2} \cdot \mathrm{P3})'$$

*5.10.8 Adders in ABEL and PLDs

ABEL supports addition (+) and subtraction (−) operators which can be applied to sets. Sets are interpreted as unsigned integers; for example, a set with n bits represents an integer in the range of 0 to 2^n–1. Subtraction is performed by negating the subtrahend and adding. Negation is performed in two’s complement; that is, the operand is complemented bit-by-bit and then 1 is added.

Table 5-53 shows an example of addition in ABEL. The set definition for SUM was made one bit wider than the addends to accommodate the carry out of the MSB; otherwise this carry would be discarded. The set definitions for the addends were extended on the left with a 0 bit to match the size of SUM.

Even though the adder program is extremely small, it takes a long time to compile and it generates a huge number of terms in the minimal two level sum of products. While SUM0 has only two product terms, subsequent terms SUM*i* have $5 \cdot 2^i$–4 terms, or 636 terms for SUM7! And the carry out (SUM8) has 2^8–1=255 product terms. Obviously, adders with more than a few bits cannot be practically realized using two levels of logic.

Table 5-53 ABEL program for an 8-bit adder.

```
module add
title 'Adder Exercise'

" Input and output pins
A7..A0, B7..B0         pin;
SUM8..SUM0             pin istype 'com';

" Set definitions
A = [0, A7..A0];
B = [0, B7..B0];
SUM = [SUM8..SUM0];

equations
SUM = A + B;

end add
```

CARRYING ON The carry out (SUM8) in our adder example has the exactly same number of product terms (255) as the "less-than" or "greater-than" output of an 8-bit comparator. This is less surprising once you realize that the carry out from the addition A+B is functionally equivalent to the expression A>$\overline{\text{B}}$.

@CARRY directive

Recognizing that larger adders and comparators are still needed in PLDs from time to time, ABEL provides an *@CARRY directive* which tells the compiler to synthesize group-ripple adder with *n* bits per group. For example, if the statement "@CARRY 1;" were included in Table 5-53, the compiler would create eight new signals for the carries out of bit positions 0 through 7. The equations for SUM1 through SUM8 would use these internal carries, essentially creating an 8-stage ripple adder with a worst-case delay of eight passes through the PLD.

If the statement "@CARRY 2;" were used, the compiler would compute carries two bits at a time, creating four new signals for carries out of bit positions 1, 3, 5, and 7. In this case, the maximum number of product terms needed for any output is still reasonable, only 7, and the worst-case delay path has just four passes through the PLD. With three bits per group (@CARRY 3;), the maximum number of product terms balloons to 28, which is impractical.

A special case that is often used in ABEL and PLDs is adding or subtracting a constant 1. This operation is used in the definition of counters, where the next state of the counter is just the current state plus 1 for an "up" counter or minus 1 for a "down" counter. The equation for bit *i* of an "up" counter can be stated very simply in words: "Complement bit *i* if counting is enabled and all of the bits lower than *i* are 1." This requires just *i*+2 product terms for any value of *i*, and can be further reduced to just one product term and an XOR gate in some PLDs, as shown in Sections 10.5.1 and 10.5.3.

*5.10.9 Adders in VHDL

Although VHDL has addition (+) and subtraction (−) operators built in, they only work with the integer, real, and physical types. They specifically do *not* work with BIT_VECTOR types or the IEEE standard type STD_LOGIC_VECTOR. Instead, standard packages define these operators.

As we explained in Section 5.9.6, the IEEE_std_logic_arith package defines two new array types, SIGNED and UNSIGNED, and a set of comparison functions for operands of type INTEGER, SIGNED, or UNSIGNED. The package also defines addition and subtraction operations for the same kinds of operands as well as STD_LOGIC and STD_ULOGIC for 1-bit operands.

The large number of overlaid addition and subtraction functions may make it less than obvious what type an addition or subtraction result will have. Normally, if any of the operands is type SIGNED, the result is SIGNED, else the result is UNSIGNED. However, if the result is assigned to a signal or variable of

Table 5-54
VHDL program for adding and subtracting 8-bit integers of various types.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vadd is
    port (
        A, B: in UNSIGNED (7 downto 0);
        C: in SIGNED (7 downto 0);
        D: in STD_LOGIC_VECTOR (7 downto 0);
        S: out UNSIGNED (8 downto 0);
        T: out SIGNED (8 downto 0);
        U: out SIGNED (7 downto 0);
        V: out STD_LOGIC_VECTOR (8 downto 0)
    );
end vadd;

architecture vadd_arch of vadd is
  begin
    S <= ('0' & A) + ('0' & B);
    T <= A + C;
    U <= C + SIGNED(D);
    V <= C - UNSIGNED(D);
end vadd_arch;
```

type STD_LOGIC_VECTOR, then the SIGNED or UNSIGNED result is converted to that type. The length of any result is normally the length of the longest operand. However, when an UNSIGNED operand is combined with a SIGNED or INTEGER operand, its length is increased by 1 to accommodate a sign bit of 0, and then the result's length is determined.

Incorporating these considerations, the VHDL program in Table 5-54 shows 8-bit additions for various operand and result types. The first result, S, is declared to be 9 bits long assuming the designer is interested in the carry from the 8-bit addition of UNSIGNED operands A and B. The concatenation operator & is used to extend A and B so that the addition function will return the carry bit in the MSB of the result.

The next result, T, is also 9 bits long, since the addition function extends the UNSIGNED operand A when combining it with the SIGNED operand C. In the third addition, an 8-bit STD_LOGIC_VECTOR D is type-converted to SIGNED and combined with C to obtain an 8-bit SIGNED result U. In the last statement, D is converted to UNSIGNED, automatically extended by one bit, and subtracted from C to produce a 9-bit result V.

Since addition and subtraction are fairly expensive in terms of the number of gates required, many VHDL synthesis engines will attempt to reuse adder blocks whenever possible. For example, Table 5-55 is a VHDL program that includes two different additions. Rather than building two adders and selecting

Table 5-55
VHDL program that allows adder sharing.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vaddshr is
    port (
        A, B, C, D: in SIGNED (7 downto 0);
        SEL: in STD_LOGIC;
        S: out SIGNED (7 downto 0)
    );
end vaddshr;

architecture vaddshr_arch of vaddshr is
begin
    S <= A + B when SEL = '1' else C + D;
end vaddshr_arch;
```

one's output with a multiplexer, the synthesis engine can build just one adder and select its inputs using multiplexers, potentially creating a smaller overall circuit.

*5.11 Combinational Multipliers

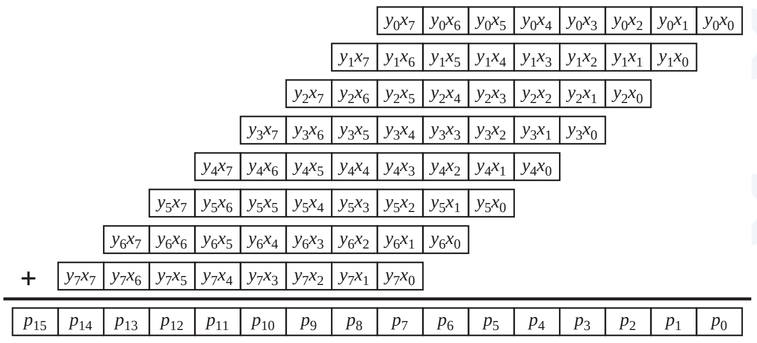
*5.11.1 Combinational Multiplier Structures

In Section 2.8, we outlined an algorithm that uses *n* shifts and adds to multiply *n*-bit binary numbers. Although the shift-and-add algorithm emulates the way that we do paper-and-pencil multiplication of decimal numbers, there is nothing inherently "sequential" or "time dependent" about multiplication. That is, given two *n*-bit input words *X* and *Y*, it is possible to write a truth table that expresses the 2*n*-bit product $P = X \cdot Y$ as a *combinational* function of *X* and *Y*. A *combinational multiplier* is a logic circuit with such a truth table.

combinational multiplier

Most approaches to combinational multiplication are based on the paper-and-pencil shift-and-add algorithm. Figure 5-96 illustrates the basic idea for an 8×8 multiplier for two unsigned integers, multiplicand $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and multiplier $Y = y_7y_6y_5y_4y_3y_2y_1y_0$. We call each row a *product component*, a shifted

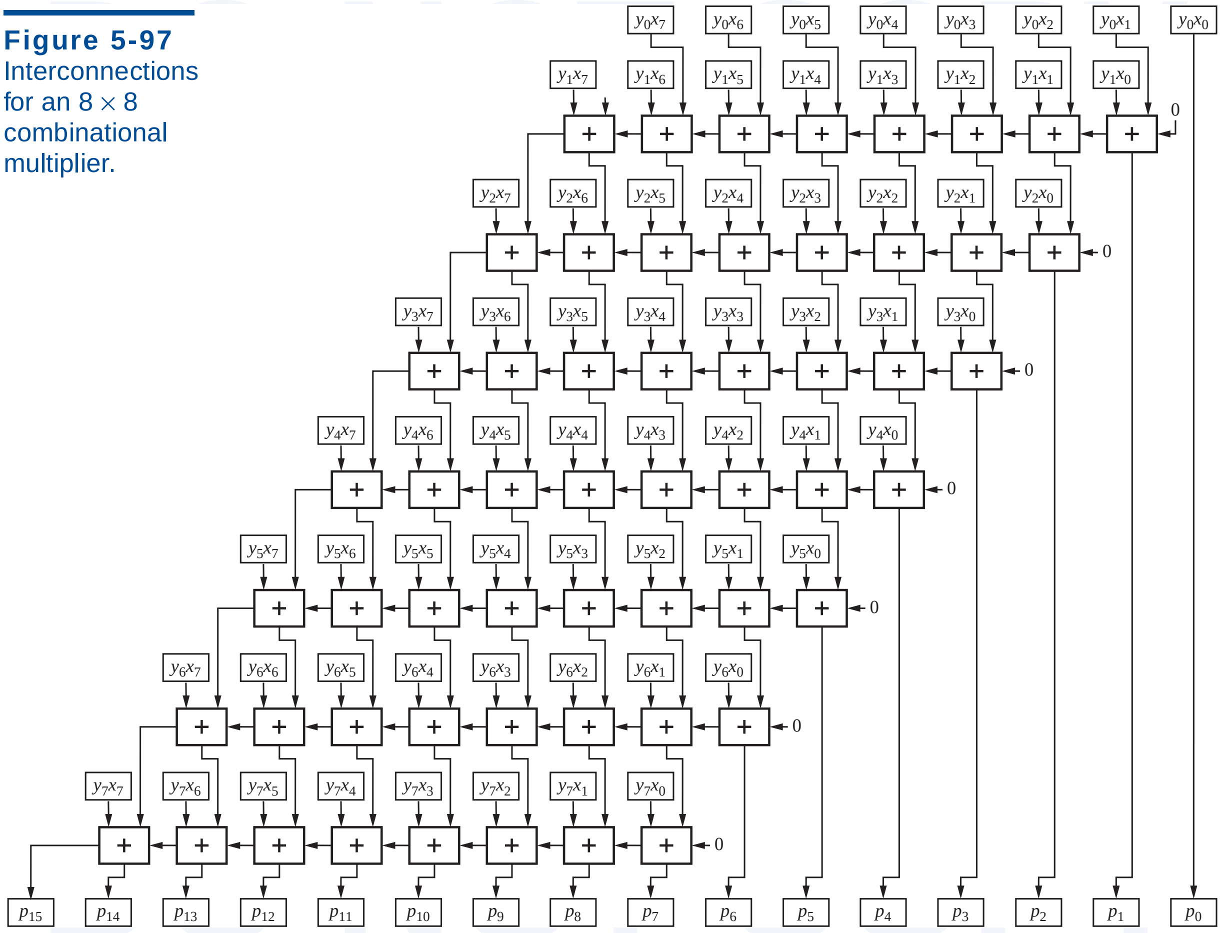
product component



y_0x_7 y_0x_6 y_0x_5 y_0x_4 y_0x_3 y_0x_2 y_0x_1 y_0x_0
y_1x_7 y_1x_6 y_1x_5 y_1x_4 y_1x_3 y_1x_2 y_1x_1 y_1x_0
y_2x_7 y_2x_6 y_2x_5 y_2x_4 y_2x_3 y_2x_2 y_2x_1 y_2x_0
y_3x_7 y_3x_6 y_3x_5 y_3x_4 y_3x_3 y_3x_2 y_3x_1 y_3x_0
y_4x_7 y_4x_6 y_4x_5 y_4x_4 y_4x_3 y_4x_2 y_4x_1 y_4x_0
y_5x_7 y_5x_6 y_5x_5 y_5x_4 y_5x_3 y_5x_2 y_5x_1 y_5x_0
y_6x_7 y_6x_6 y_6x_5 y_6x_4 y_6x_3 y_6x_2 y_6x_1 y_6x_0
+ y_7x_7 y_7x_6 y_7x_5 y_7x_4 y_7x_3 y_7x_2 y_7x_1 y_7x_0
p_{15} p_{14} p_{13} p_{12} p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 5-96
Partial products in an 8×8 multiplier.

Figure 5-97
Interconnections for an 8 × 8 combinational multiplier.

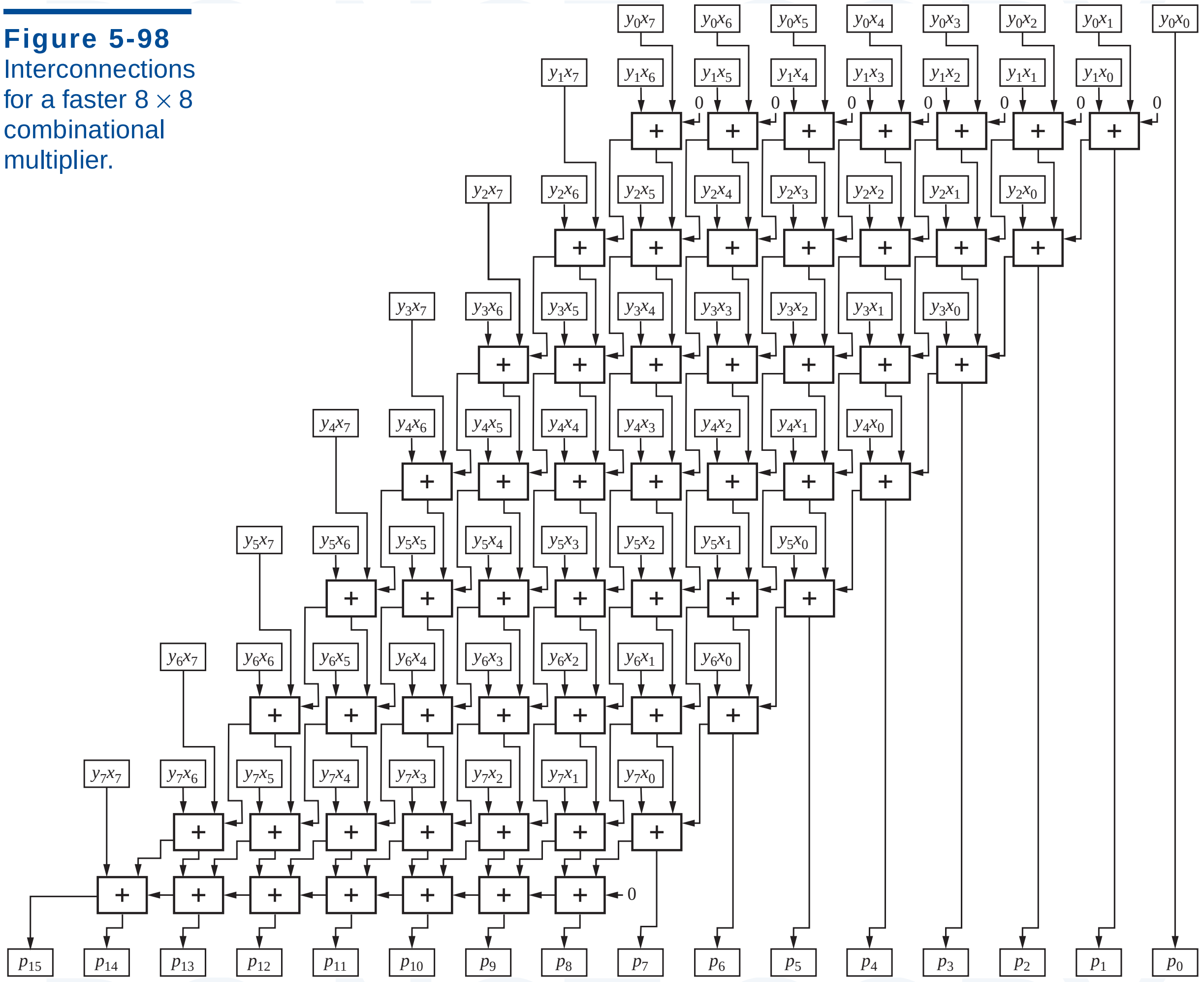


y_0x_7 y_0x_6 y_0x_5 y_0x_4 y_0x_3 y_0x_2 y_0x_1 y_0x_0
y_1x_7 y_1x_6 y_1x_5 y_1x_4 y_1x_3 y_1x_2 y_1x_1 y_1x_0
y_2x_7 y_2x_6 y_2x_5 y_2x_4 y_2x_3 y_2x_2 y_2x_1 y_2x_0
y_3x_7 y_3x_6 y_3x_5 y_3x_4 y_3x_3 y_3x_2 y_3x_1 y_3x_0
y_4x_7 y_4x_6 y_4x_5 y_4x_4 y_4x_3 y_4x_2 y_4x_1 y_4x_0
y_5x_7 y_5x_6 y_5x_5 y_5x_4 y_5x_3 y_5x_2 y_5x_1 y_5x_0
y_6x_7 y_6x_6 y_6x_5 y_6x_4 y_6x_3 y_6x_2 y_6x_1 y_6x_0
y_7x_7 y_7x_6 y_7x_5 y_7x_4 y_7x_3 y_7x_2 y_7x_1 y_7x_0
+
0
p_{15} p_{14} p_{13} p_{12} p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit. Each small box represents one product-component bit y_ix_j, the logical AND of multiplier bit y_i and multiplicand bit x_j. The product $P = p_{15}p_{14}\ldots p_2p_1p_0$ has 16 bits and is obtained by adding together all the product components.

Figure 5-97 shows one way to add up the product components. Here, the product-component bits have been spread out to make space, and each "+" box is a full adder equivalent to Figure 5-85(c) on page 391. The carries in each row of full adders are connected to make an 8-bit ripple adder. Thus, the first ripple adder combines the first two product components to product the first partial product, as defined in Section 2.8. Subsequent adders combine each partial product with the next product component.

It is interesting to study the propagation delay of the circuit in Figure 5-97. In the worst case, the inputs to the least significant adder (y_0x_1 and y_1x_0) can affect the MSB of the product (p_{15}). If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst case



y_0x_7 y_0x_6 y_0x_5 y_0x_4 y_0x_3 y_0x_2 y_0x_1 y_0x_0
y_1x_7 y_1x_6 y_1x_5 y_1x_4 y_1x_3 y_1x_2 y_1x_1 y_1x_0
y_2x_7 y_2x_6 y_2x_5 y_2x_4 y_2x_3 y_2x_2 y_2x_1 y_2x_0
y_3x_7 y_3x_6 y_3x_5 y_3x_4 y_3x_3 y_3x_2 y_3x_1 y_3x_0
y_4x_7 y_4x_6 y_4x_5 y_4x_4 y_4x_3 y_4x_2 y_4x_1 y_4x_0
y_5x_7 y_5x_6 y_5x_5 y_5x_4 y_5x_3 y_5x_2 y_5x_1 y_5x_0
y_6x_7 y_6x_6 y_6x_5 y_6x_4 y_6x_3 y_6x_2 y_6x_1 y_6x_0
y_7x_7 y_7x_6 y_7x_5 y_7x_4 y_7x_3 y_7x_2 y_7x_1 y_7x_0
+
0
p_{15} p_{14} p_{13} p_{12} p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 5-98
Interconnections for a faster 8 × 8 combinational multiplier.

path goes through 20 adders and its delay is $20t_{pd}$. If the delays are different, then the answer depends on the relative delays; see Exercise \exref{xxxx}.

sequential multiplier

Sequential multipliers use a single adder and a register to accumulate the partial products. The partial-product register is initialized to the first product component, and for an $n \times n$-bit multiplication, $n-1$ steps are taken and the adder is used $n-1$ times, once for each of the remaining $n-1$ product components to be added to the partial-product register.

carry-save addition

Some sequential multipliers use a trick called *carry-save addition* to speed up multiplication. The idea is to break the carry chain of the ripple adder to shorten the delay of each addition. This is done by applying the carry output from bit i during step j to the carry input for bit $i+1$ during the *next* step, $j+1$. After the last product component is added, one more step is needed in which the carries are hooked up in the usual way and allowed to ripple from the least to the most significant bit.

The combinational equivalent of an 8×8 multiplier using carry-save addition is shown in Figure 5-98. Notice that the carry out of each full adder in the first seven rows is connected to an input of an adder *below* it. Carries in the eighth row of full adders are connected to create a conventional ripple adder. Although this adder uses exactly the same amount of logic as the previous one (64 2-input AND gates and 56 full adders), its propagation delay is substantially shorter. Its worst-case delay path goes through only 14 full adders. The delay can be further improved by using a carry lookahead adder for the last row.

The regular structure of combinational multipliers make them ideal for VLSI and ASIC realization. The importance of fast multiplication in microprocessors, digital video, and many other applications has led to much study and development of even better structures and circuits for combinational multipliers; see the References.

*5.11.2 Multiplication in ABEL and PLDs

ABEL provides a multiplication operator `*`, but it can be used only with individual signals, numbers, or special constants, not with sets. Thus, ABEL cannot synthesize a multiplier circuit from a single equation like “`P = X*Y`.”

Still, you can use ABEL to specify a combinational multiplier if you break it down into smaller pieces. For example, Table 5-56 shows the design of a 4×4 unsigned multiplier following the same general structure as Figure 5-96 on page page 406. Expressions are used to define the four product components, `PC1`, `PC2`, `PC3`, and `PC4`, which are then added in the `equations` section of the program. This does not generate an array of full adders as in Figure 5-97 or 5-98. Rather, the ABEL compiler will dutifully crunch the addition equation to pro-

```
module mul4x4
title '4x4 Combinational Multiplier'

X3..X0, Y3..Y0 pin;  " multiplicand, multiplier
P7..P0         pin istype 'com';    " product

P = [P7..P0];
PC1 = Y0 & [0, 0, 0, 0,X3,X2,X1,X0];
PC2 = Y1 & [0, 0, 0,X3,X2,X1,X0, 0];
PC3 = Y2 & [0, 0,X3,X2,X1,X0, 0, 0];
PC4 = Y3 & [0,X3,X2,X1,X0, 0, 0, 0];

equations
P = PC1 + PC2 + PC3 + PC4;

end mul4x4
```

Table 5-56 ABEL program for a 4×4 combinational multiplier.

duce a minimal sum for each of the eight product output bits. Surprisingly, the worst-case output, P4, has only 36 product terms, a little high but certainly realizable in two passes through a PLD.

*5.11.3 Multiplication in VHDL

VHDL is rich enough to express multiplication in a number of different ways; we'll save the best for last.

Table 5-57 is a behavioral VHDL program that mimics the multiplier structure of Figure 5-98. In order to represent the internal signals in the figure, the program defines a new data type, `array8x8`, which is a two-dimensional array of `STD_LOGIC` (recall that `STD_LOGIC_VECTOR` is a one-dimensional array of `STD_LOGIC`). Variable `PC` is declared as a such an array to hold the product-component bits, and variables `PCS` and `PCC` are similar arrays to hold the sum and carry outputs of the main array of full adders. One-dimensional arrays `RAS` and `RAC` hold the sum and carry outputs of the ripple adder. Figure 5-99 shows the variable naming and numbering scheme. Integer variables `i` and `j` are used as loop indices for rows and columns, respectively.

The program attempts to illustrate the logic gates that would be used in a faithful realization of Figure 5-98, even though a synthesizer could legitimately create quite a different structure from this behavioral program. If you want to control the structure, then you must use structural VHDL, as we'll show later.

In the program, the first, nested `for` statement performs 64 `AND` operations to obtain the product-component bits. The next `for` loop initializes boundary conditions at the top of the multiplier, using the notion of row-0 "virtual" full adders, not shown in the figure, whose sum outputs equal the first row of `PC` bits

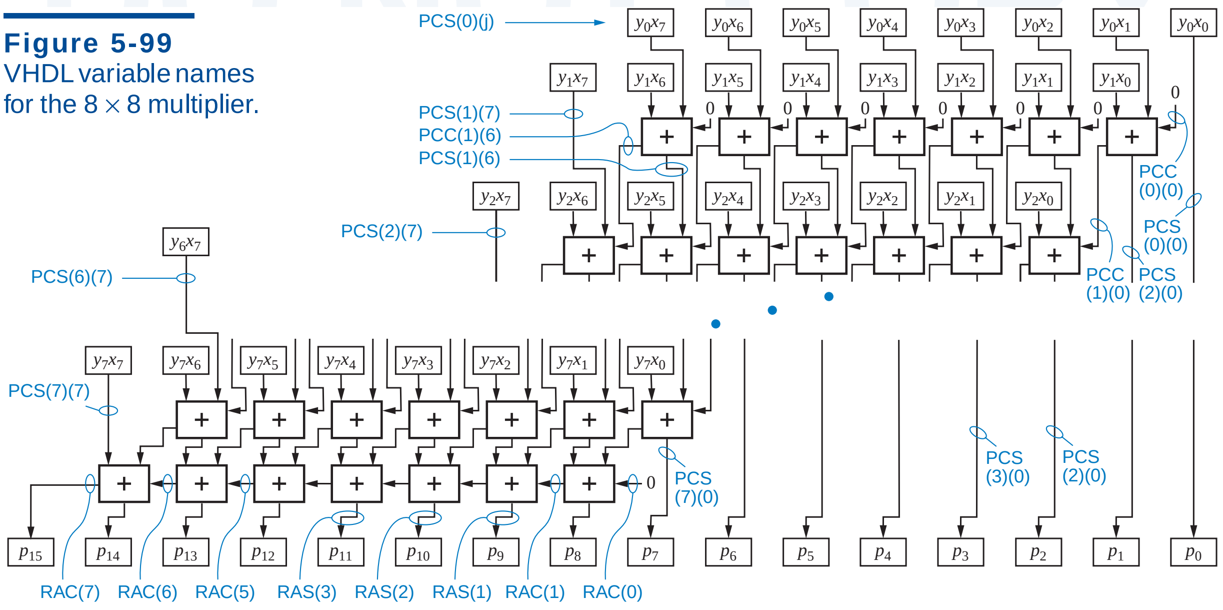


Figure 5-99 VHDL variable names for the 8 × 8 multiplier.

Table 5-57
Behavioral VHDL program for an 8×8 combinational multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity vmul8x8p is
    port ( X: in STD_LOGIC_VECTOR (7 downto 0);
           Y: in STD_LOGIC_VECTOR (7 downto 0);
           P: out STD_LOGIC_VECTOR (15 downto 0) );
end vmul8x8p;

architecture vmul8x8p_arch of vmul8x8p is
function MAJ (I1, I2, I3: STD_LOGIC) return STD_LOGIC is
  begin
    return ((I1 and I2) or (I1 and I3) or (I2 and I3));
  end MAJ;
begin
process (X, Y)
type array8x8 is array (0 to 7) of STD_LOGIC_VECTOR (7 downto 0);
variable PC: array8x8;     -- product component bits
variable PCS: array8x8;    -- full-adder sum bits
variable PCC: array8x8;    -- full-adder carry output bits
variable RAS, RAC: STD_LOGIC_VECTOR (7 downto 0); -- ripple adder sum
  begin                                           --   and carry bits
    for i in 0 to 7 loop for j in 0 to 7 loop
        PC(i)(j) := Y(i) and X(j); -- compute product component bits
    end loop;  end loop;
    for j in 0 to 7 loop
      PCS(0)(j) := PC(0)(j);  -- initialize first-row "virtual"
      PCC(0)(j) := '0';       --   adders (not shown in figure)
    end loop;
    for i in 1 to 7 loop      -- do all full adders except last row
      for j in 0 to 6 loop
        PCS(i)(j) := PC(i)(j) xor PCS(i-1)(j+1) xor PCC(i-1)(j);
        PCC(i)(j) := MAJ(PC(i)(j), PCS(i-1)(j+1), PCC(i-1)(j));
        PCS(i)(7) := PC(i)(7); -- leftmost "virtual" adder sum output
      end loop;
    end loop;
    RAC(0) := '0';
    for i in 0 to 6 loop  -- final ripple adder
      RAS(i) := PCS(7)(i+1) xor PCC(7)(i) xor RAC(i);
      RAC(i+1) := MAJ(PCS(7)(i+1), PCC(7)(i), RAC(i));
    end loop;
    for i in 0 to 7 loop
      P(i) <= PCS(i)(0);  -- first 8 product bits from full-adder sums
    end loop;
    for i in 8 to 14 loop
      P(i) <= RAS(i-8);   -- next 7 bits from ripple-adder sums
    end loop;
    P(15) <= RAC(7);      -- last bit from ripple-adder carry
  end process;
end vmul8x8p_arch;
```

SIGNALS VS. VARIABLES

Variables are used rather than signals in the process in Table 5-57 to make simulation run faster. Variables are faster because the simulator keeps track of their values only when the process is running. Because variable values are assigned sequentially, the process in Table 5-57 is carefully written to compute values in the proper order. That is, a variable cannot be used until a value has been assigned to it.

Signals, on the other hand, have a value at all times. When a signal value is changed in a process, the simulator schedules a future event in its event list for the value change. If the signal appears on the right-hand side of an assignment statement in the process, then the signal must also be included in the process' sensitivity list. If a signal value changes, the process will then execute again, and keep repeating until all of the signals in the sensitivity list are stable.

In Table 5-57, if you wanted to observe internal values or timing during simulation, you could change all the variables (except `i` and `j`) to signals and include them in the sensitivity list. To make the program syntactically correct, you would also have to move the `type` and `signal` declarations to just after the architecture statement, and change all of the "`:=`" assignments to "`<=`".

Suppose that after making the changes above, you also reversed the order of the indices in the `for` loops (e.g., "`7 downto 0`" instead of "`0 to 7`"). The program would still work. However, dozens of repetitions of the process would be required for each input change in `X` or `Y`, because the signal changes in this circuit propagate from the lowest index to the highest.

While the choice of signals vs. variables affects the speed of simulation, with most VHDL synthesis engines it does not affect the results of synthesis.

and whose carry outputs are 0. The third, nested `for` loop corresponds to the main array of adders in Figure 5-98, all except the last row, which is handled by the fourth `for` loop. The last two `for` loops assign the appropriate adder outputs to the multiplier output signals.

ON THE THRESHOLD OF A DREAM

A three-input "majority function," `MAJ`, is defined at the beginning of Table 5-57 and is subsequently used to compute carry outputs. An *n*-input *majority function* produces a 1 output if the majority of its inputs are 1, two out of three in the case of a 3-input majority function. (If *n* is even, *n*/2+1 inputs must be 1.)

Over thirty years ago, there was substantial academic interest in a more general class of *n*-input *threshold functions* which produce a 1 output if *k* or more of their inputs are 1. Besides providing full employment for logic theoreticians, threshold functions could realize many logic functions with a smaller number of elements than could a conventional AND/OR realization. For example, an adder's carry function requires three AND gates and one OR gate, but just one three-input threshold gate.

(Un)fortunately, an economical technology never emerged for threshold gates, and they remain, for now, an academic curiosity.

Table 5-58
Structural VHDL architecture for an 8×8 combinational multiplier.

```
architecture vmul8x8s_arch of vmul8x8s is
component AND2
  port( I0, I1: in STD_LOGIC;
        O: out STD_LOGIC );
end component;
component XOR3
  port( I0, I1, I2: in STD_LOGIC;
        O: out STD_LOGIC );
end component;
component MAJ   -- Majority function, O = I0*I1 + I0*I2 + I1*I2
  port( I0, I1, I2: in STD_LOGIC;
        O: out STD_LOGIC );
end component;

type array8x8 is array (0 to 7) of STD_LOGIC_VECTOR (7 downto 0);
signal PC: array8x8;     -- product-component bits
signal PCS: array8x8;    -- full-adder sum bits
signal PCC: array8x8;    -- full-adder carry output bits
signal RAS, RAC: STD_LOGIC_VECTOR (7 downto 0); -- sum, carry
begin
  g1: for i in 0 to 7 generate    -- product-component bits
    g2: for j in 0 to 7 generate
        U1: AND2 port map (Y(i), X(j), PC(i)(j));
    end generate;
  end generate;
  g3: for j in 0 to 7 generate
    PCS(0)(j) <= PC(0)(j);  -- initialize first-row "virtual" adders
    PCC(0)(j) <= '0';
  end generate;
  g4: for i in 1 to 7 generate   -- do full adders except the last row
    g5: for j in 0 to 6 generate
      U2: XOR3 port map (PC(i)(j),PCS(i-1)(j+1),PCC(i-1)(j),PCS(i)(j));
      U3: MAJ  port map (PC(i)(j),PCS(i-1)(j+1),PCC(i-1)(j),PCC(i)(j));
      PCS(i)(7) <= PC(i)(7); -- leftmost "virtual" adder sum output
    end generate;
  end generate;
  RAC(0) <= '0';
  g6: for i in 0 to 6 generate  -- final ripple adder
    U7: XOR3 port map (PCS(7)(i+1), PCC(7)(i), RAC(i), RAS(i));
    U3: MAJ  port map (PCS(7)(i+1), PCC(7)(i), RAC(i), RAC(i+1));
  end generate;
  g7: for i in 0 to 7 generate
   P(i) <= PCS(i)(0);  -- get first 8 product bits from full-adder sums
  end generate;
  g8: for i in 8 to 14 generate
    P(i) <= RAS(i-8);   -- get next 7 bits from ripple-adder sums
  end generate;
  P(15) <= RAC(7);      -- get last bit from ripple-adder carry
end vmul8x8s_arch;
```

The program in Table 5-57 can be modified to use structural VHDL as shown in Table 5-58. This approach gives the designer complete control over the circuit structure that is synthesized, as might be desired in an ASIC realization. The program assumes that the architectures for AND2, XOR3, and MAJ3 have been defined elsewhere, for example, in an ASIC library.

generate statement

This program makes good use of the *generate statement* to create the arrays of components used in the multiplier. The generate statement must have a label, and similar to a for-loop statement, it specifies an iteration scheme to control the repetition of the enclosed statements. Within for-generate, the enclosed statements can include any concurrent statements, IF-THEN-ELSE statements, and additional levels of looping constructs. Sometimes generate statements are combined with IF-THEN-ELSE to produce a kind of conditional compilation capability

Well, we said we'd save the best for last, and here it is. The IEEE std_logic_arith library that we introduced in Section 5.9.6 defines multiplication functions for SIGNED and UNSIGNED types, and overlays these functions onto the "*" operator. Thus, the program in Table 5-59 can multiply unsigned numbers with a simple one-line assignment statement. Within the IEEE library, the multiplication function is defined behaviorally, using the shift-and-add algorithm. We could have showed you this approach at the beginning of this subsection, but then you wouldn't have read the rest of it, would you?

Table 5-59 Truly behavioral VHDL program for an 8×8 combinational multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity vmul8x8i is
    port (
        X: in UNSIGNED (7 downto 0);
        Y: in UNSIGNED (7 downto 0);
        P: out UNSIGNED (15 downto 0)
    );
end vmul8x8i;

architecture vmul8x8i_arch of vmul8x8i is
begin
  P <= X * Y;
end vmul8x8i_arch;
```

References

Digital designers who want to write better should start by reading the classic *Elements of Style*, 3rd ed., by William Strunk, Jr. and E. B. White (Allyn & Bacon, 1979). Another book on writing style, especially for nerds, is *Effective*

SYNTHESIS OF BEHAVIORAL DESIGNS

You’ve probably heard that compilers for high-level programming languages like C usually generate better code than people do writing in assembly language, even with “hand-tweaking.” Most digital designers hope that compilers for behavioral HDLs will also some day produce results superior to a typical hand-tweaked design, be it a schematic or structural VHDL. Better compilers won’t put the designers out of work, they will simply allow them to tackle bigger designs.

We’re not quite there yet. However, the more advanced synthesis engines do include some nice optimizations for commonly used behavioral structures. For example, I have to admit that the FPGA synthesis engine that I used to test the VHDL programs in this subsection produced just as fast a multiplier from Table 5-59 as it did from any of the more detailed architectures!

Writing for Engineers, Managers, and Scientists, 2nd ed., by H. J. Tichy (Wiley, 1988). Plus two new books from Amazon.

The ANSIIEEE standard for logic symbols is Std 91-1984, *IEEE Standard Graphic Symbols for Logic Functions*. Another standard of interest to logic designers is ANSI/IEEE 991-1986, *Logic Circuit Diagrams*. These two standards and ten others, including standard symbols for 10-inch gongs and maid’s-signal plugs, can be found in one handy, five-pound reference, *Electrical and Electronics Graphic and Letter Symbols and Reference Designations Standards Collection Electrical and Electronics Graphics Symbols and Reference Designations* published by the IEEE in 1996 (`www.ieee.org`).

Real logic devices are described in data sheets and data books published by the manufacturers. Updated editions of data books used to be published every few years, but in recent years the trend has been to minimize or eliminate the hardcopy editions and instead to publish up-to-date information on the web. Two of the largest suppliers with the most comprehensive sites are Texas Instruments (`www.ti.com`) and Motorola (`www.mot.com`).

For a given logic family such as 74ALS, all manufacturers list generally equivalent specifications, so you can get by with just one data book per family. Some specifications, especially timing, may vary slightly between manufacturers, so when timing is tight it’s best to check a couple of different sources and use the worst case. That’s a *lot* easier than convincing your manufacturing department to buy a component only from a single supplier.

The first PAL devices were invented in 1978 by John Birkner at Monolithic Memories, Inc. (MMI). Birkner earned U.S. patent number 4,124,899 for his invention, and MMI rewarded him by buying him a brand new Porsche! Seeing the value in this technology (PALs, not Porsches), Advanced Micro Devices (AMD) acquired MMI in the early 1980s and remained a leading developer and supplier of new PLDs and CPLDs. In 1997, AMD spun off its PLD operations to form Vantis Corporation.

Some of the best resources for learning about PLD-based design are provided the PLD manufacturers. For example, Vantis publishes the *1999 Vantis Data Book (*Sunnyvale, CA 94088, 1999), which contains information on their PLDs, CPLDs, and FPGAs. Individual data sheets and application notes are readily downloadable from their web site (`www.vantis.com`). Similarly, GAL inventor Lattice Semiconductor has a comprehensive *Lattice Data Book* (Hillsboro, OR 97124, 1999) and web site (`www.latticesemi.com`).

A much more detailed discussion of the internal operation of LSI and VLSI devices, including PLDs, ROMs, and RAMs, can be found in electronics texts such as *Microelectronics*, 2nd ed., by J.Millman and A.Grabel (McGraw-Hill, 1987) and *VLSI Engineering* by Thomas E. Dillinger (Prentice Hall, 1988). Additional PLD references are cited at the end of \chapref{SeqPLDs}.

On the technical side of digital design, lots of textbooks cover digital design principles, but only a few cover practical aspects of design. A classic is *Digital Design with Standard MSI and LSI* by Thomas R. Blakeslee, 2nd ed. (Wiley, 1979), which includes chapters on everything from structured combinational design with MSI and LSI to “the social consequences of engineering.” A more recent, excellent short book focusing on digital design practices is *The Well-Tempered Digital Design* by Robert B. Seidensticker (Addison-Wesley, 1986). It contains hundreds of readily accessible digital-design “proverbs” in areas ranging from high-level design philosophy to manufacturability. Another easy-reading, practical, and “fun” book on analog and digital design is Clive Maxfield’s *Bebop to the Boolean Boogie* (LLH Technology Publishing, 1997; for a good time, also visit `www.maxmon.com`).

Drill Problems

5.1 Give three examples of combinational logic circuits that require *billions and billions* of rows to describe in a truth table. For each circuit, describe the circuit’s inputs and output(s), and indicate exactly how many rows the truth table contains; you need not write out the truth table. (*Hint:* You can find several such circuits right in this chapter.)

5.2 Draw the DeMorgan equivalent symbol for a 74x30 8-input NAND gate.

5.3 Draw the DeMorgan equivalent symbol for a 74x27 3-input NOR gate.

5.4 What’s wrong with the signal name “READY′”?

5.5 You may find it annoying to have to keep track of the active levels of all the signals in a logic circuit. Why not use only noninverting gates, so all signals are active high?

5.6 True or false: In bubble-to-bubble logic design, outputs with a bubble can be connected only to inputs with a bubble.

5.7 A digital communication system is being designed with twelve identical network ports. Which type of schematic structure is probably most appropriate for the design?

5.8 Determine the exact maximum propagation delay from IN to OUT of the circuit in Figure X5.8 for both LOW-to-HIGH and HIGH-to-LOW transitions, using the timing information given in Table 5-2. Repeat, using a single worst-case delay number for each gate and compare and comment on your results.

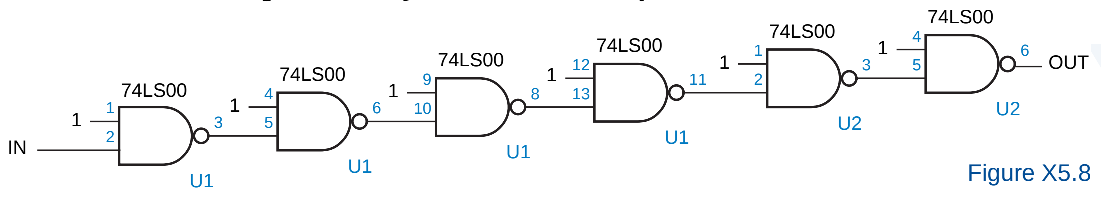


Figure X5.8

5.9 Repeat Drill 5.8, substituting 74HCT00s for the 74LS00s.

5.10 Repeat Drill 5.8, substituting 74LS08s for the 74LS00s.

5.11 Repeat Drill 5.8, substituting 74AHCT02s for the 74LS00s, using constant 0 instead of constant 1 inputs, and using typical rather than maximum timing.

5.12 Estimate the minimum propagation delay from IN to OUT for the circuit shown in Figure X5.12. Justify your answer.

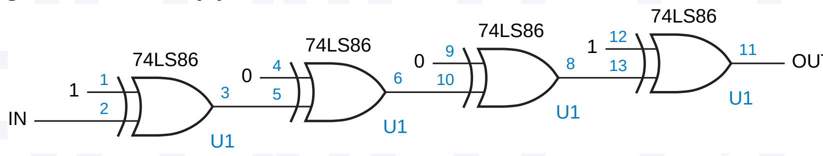


Figure X5.12

5.13 Determine the exact maximum propagation delay from IN to OUT of the circuit in Figure X5.12 for both LOW-to-HIGH and HIGH-to-LOW transitions, using the timing information given in Table 5-2. Repeat, using a single worst-case delay number for each gate and compare and comment on your results.

5.14 Repeat Drill 5.13, substituting 74HCT86s for the 74LS86s.

5.15 Which would expect to be faster, a decoder with active-high outputs or one with active-low outputs?

5.16 Using the information in Table 5-3 for 74LS components, determine the maximum propagation delay from any input to any output in the 5-to-32 decoder circuit of Figure 5-39. You may use the "worst-case" analysis method.

5.17 Repeat Drill 5.16, performing a detailed analysis for each transition direction, and compare your results.

5.18 Show how to build each of the following single- or multiple-output logic functions using one or more 74x138 or 74x139 binary decoders and NAND gates. (Hint: Each realization should be equivalent to a sum of minterms.)

(a) $F = \Sigma_{X,Y,Z}(2,4,7)$ (b) $F = \prod_{A,B,C}(3,4,5,6,7)$

(c) $F = \Sigma_{A,B,C,D}(2,4,6,14)$ (d) $F = \Sigma_{W,X,Y,Z}(0,1,2,3,5,7,11,13)$

(e) $F = \Sigma_{W,X,Y}(1,3,5,6)$ (f) $F = \Sigma_{A,B,C}(0,4,6)$

$G = \Sigma_{W,X,Y}(2,3,4,7)$ $G = \Sigma_{C,D,E}(1,2)$

5.19 Draw the digits created by a 74x49 seven-segment decoder for the nondecimal inputs 1010 through 1111.

5.20 Starting with the logic diagram for the 74x148 priority encoder, write logic equations for its A2_L, A1_L, and A0_L outputs. How do they differ from the "generic" equations given in Section 5.5.1?

5.21 What's terribly wrong with the circuit in Figure X5.21? Suggest a change that eliminates the terrible problem.

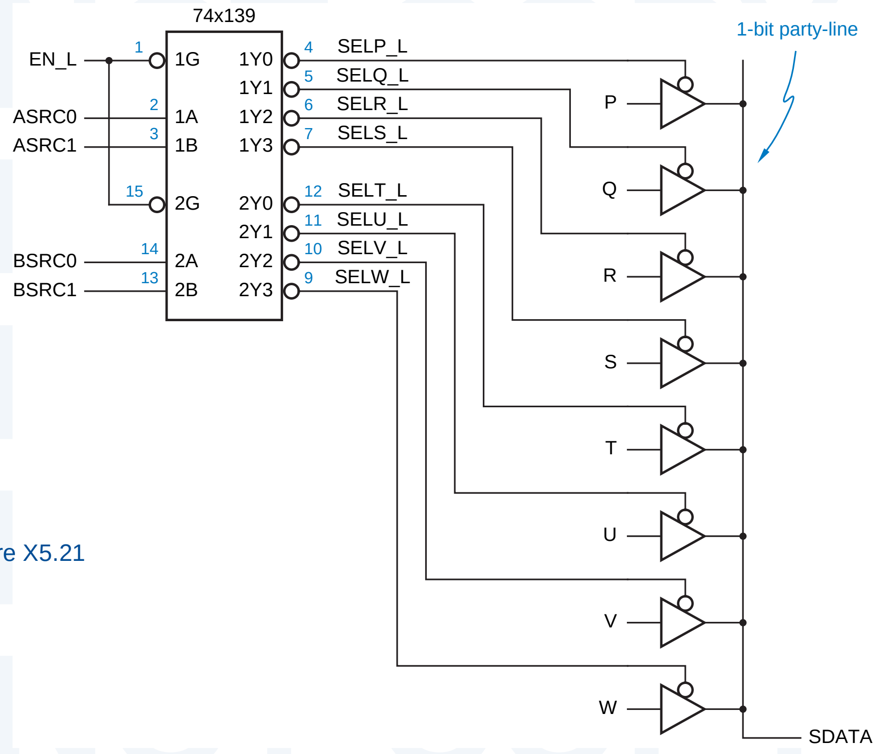


Figure X5.21

5.22 Using the information in Tables 5-2 and 5-3 for 74LS components, determine the maximum propagation delay from any input to any output in the 32-to-1 multiplexer circuit of Figure 5-65. You may use the "worst-case" analysis method.

5.23 Repeat Exercise 5.22 using 74HCT components.

5.24 An *n*-input parity tree can be built with XOR gates in the style of Figure 5-73(a). Under what circumstances does a similar *n*-input parity tree built using XNOR gates perform exactly the same function?

5.25 Using the information in Tables 5-2 and 5-3 for 74LS components, determine the maximum propagation delay from the DU bus to the DC bus in the error-correction circuit of Figure 5-76. You may use the "worst-case" analysis method.

5.26 Repeat Exercise 5.25 using 74HCT components.

5.27 Starting with the equations given in Section 5.9.4, write a complete logic expression for the ALTBOUT output of the 74x85.

5.28 Starting with the logic diagram for the 74x682, write a logic expression for the PGTQ_L output in terms of the inputs.

5.29 Write an algebraic expression for s_2, the third sum bit of a binary adder, as a function of inputs x_0, x_1, x_2, y_0, y_1, and y_2. Assume that $c_0 = 0$, and do not attempt to "multiply out" or minimize the expression.

5.30 Using the information in Table 5-3 for 74LS components, determine the maximum propagation delay from any input to any output of the 16-bit group ripple adder of Figure 5-91. You may use the "worst-case" analysis method.

Exercises

5.31 A possible definition of a BUT gate (Exercise 4.45) is "Y1 is 1 if A1 and B1 are 1 *but* either A2 or B2 is 0; Y2 is defined symmetrically." Write the truth table and find minimal sum-of-products expressions for the BUT-gate outputs. Draw the logic diagram for a NAND-NAND circuit for the expressions, assuming that only uncomplemented inputs are available. You may use gates from 74HCT00, '04, '10, '20, and '30 packages.

5.32 Find a gate-level design for the BUT gate defined in Exercise 5.31 that uses a minimum number of transistors when realized in CMOS. You may use gates from 74HCT00, '02, '04, '10, '20, and '30 packages. Write the output expressions (which need not be two-level sums-of-products), and draw the logic diagram.

5.33 For each circuit in the two preceding exercises, compute the worst-case delay from input to output, using the delay numbers for 74HCT components in Table 5-2. Compare the cost (number of transistors), speed, and input loading of the two designs. Which is better?

5.34 Butify the function $F = \Sigma_{W,X,Y,Z}(3,7,11,12,13,14)$. That is, show how to perform F with a single BUT gate as defined in Exercise 5.31 and a single 2-input OR gate. *butification*

5.35 Design a 1-out-of-4 checker with four inputs, A, B, C, D, and a single output ERR. The output should be 1 if two or more of the inputs are 1, and 0 if no input or one input is 1. Use SSI parts from Figure 5-18, and try to minimize the number of gates required. (*Hint:* It can be done with seven two-input inverting gates.)

5.36 Suppose that a 74LS138 decoder is connected so that all enable inputs are asserted and C B A = 101. Using the information in Table 5-3 and the '138 internal logic diagram, determine the propagation delay from input to all relevant outputs for each possible single-input change. (*Hint:* There are a total of nine delay numbers, since a change on A, B, or C affects two outputs, and a change on any of the three enable inputs affects one output.)

5.37 Suppose that you are asked to design a new component, a decimal decoder that is optimized for applications in which only decimal input combinations are expected to occur. How can the cost of such a decoder be minimized compared to one that is simply a 4-to-16 decoder with six outputs removed? Write the logic equations for all ten outputs of the minimized decoder, assuming active-high inputs and outputs and no enable inputs.

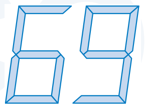
5.38 How many Karnaugh maps would be required to work Exercise 5.37 using the formal multiple-output minimization procedure described in Section 4.3.8?

5.39 Suppose that a system requires a 5-to-32 binary decoder with a single active-low enable input, a design similar to Figure 5-39. With the EN1 input pulled HIGH, either the EN2_L or the EN3_L input in the figure could be used as the enable, with the other input grounded. Discuss the pros and cons of using EN2_L versus EN3_L.

5.40 Determine whether the circuits driving the a, b, and c outputs of the 74x49 seven-segment decoder correspond to minimal product-of-sums expressions for these segments, assuming that the nondecimal input combinations are “don’t cares” and BI = 1.

5.41 Redesign the MSI 74x49 seven-segment decoder so that the digits 6 and 9 have tails as shown in Figure X5.41. Are any of the digit patterns for nondecimal inputs 1010 through 1111 affected by your redesign?

Figure X5.41



5.42 Starting with the ABEL program in Table 5-21, write a program for a seven-segment decoder with the following enhancements:

- The outputs are all active low.
- Two new inputs, ENHEX and ERRDET, control the decoding of the segment outputs.
- If ENHEX = 0, the outputs match the behavior of a 74x49.
- If ENHEX = 1, then the outputs for digits 6 and 9 have tails, and the outputs for digits A–F are controlled by ERRDET.
- If ENHEX = 1 and ERRDET = 0, then the outputs for digits A–F look like the letters A–F, as in the original program.
- If ENHEX = 1 and ERRDET = 1, then the output for digits A–F looks like the letter S.

5.43 A famous logic designer decided to quit teaching and make a fortune by fabricating huge quantities of the MSI circuit shown in Figure X5.47.

5-44 (a)Label the inputs and outputs of the circuit with appropriate signal names, including active-level indications.

5-45 (b)What does the circuit do? Be specific and account for all inputs and outputs.

5-46 (c)Draw the MSI logic symbol that would go on the data sheet of this wonderful device.

5-47 (d)With what standard MSI parts does the new part compete? Do you think it would be successful in the MSI marketplace?

5.48 An FCT three-state buffer drives ten FCT inputs and a 4.7-KΩ pull-up resistor to 5.0 V. When the output changes from LOW to Hi-Z, estimate how long it takes for the FCT inputs to see the output as HIGH. State any assumptions that you make.

5.49 On a three-state bus, ten FCT three-state buffers are driving ten FCT inputs and a 4.7-KΩ pull-up resistor to 5.0 V. Assuming that no other devices are driving the bus, estimate how long the bus signal remains at a valid logic level when an active output enters the Hi-Z state. State any assumptions that you make.

5.50 Design a 10-to-4 encoder with inputs in the 1-out-of-10 code and outputs in BCD.

5.51 Draw the logic diagram for a 16-to-4 encoder using just four 8-input NAND gates. What are the active levels of the inputs and outputs in your design?

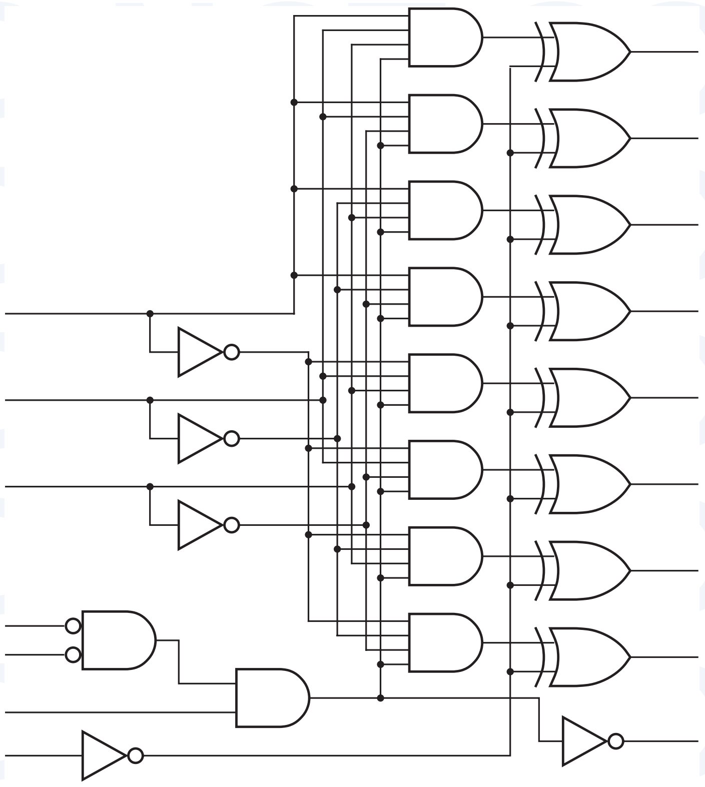


Figure X5.47

5.52 Draw the logic diagram for a circuit that uses the 74x148 to resolve priority among eight active-high inputs, I0–I7, where I7 has the highest priority. The circuit should produce active-high address outputs A2–A0 to indicate the number of the highest-priority asserted input. If no input is asserted, then A2–A0 should be 111 and an IDLE output should be asserted. You may use discrete gates in addition to the ’148. Be sure to name all signals with the proper active levels.

5.53 Draw the logic diagram for a circuit that resolves priority among eight active-low inputs, I0_L–I7_L, where I0_L has the highest priority. The circuit should produce active-high address outputs A2–A0 to indicate the number of the highest-priority asserted input. If at least one input is asserted, then an AVALID output should be asserted. Be sure to name all signals with the proper active levels. This circuit can be built with a single 74x148 and no other gates.

5.54 A purpose of Exercise 5.53 was to demonstrate that it is not always possible to maintain consistency in active-level notation unless you are willing to define alternate logic symbols for MSI parts that can be used in different ways. Define an alternate symbol for the 74x148 that provides this consistency in Exercise 5.53.

5.55 Design a combinational circuit with eight active-low request inputs, R0_L–R7_L, and eight outputs, A2–A0, AVALID, B2–B0, and BVALID. The R0_L–R7_L inputs and A2–A0 and AVALID outputs are defined as in Exercise 5.53. The B2–B0 and BVALID outputs identify the second-highest priority request input that is asserted.

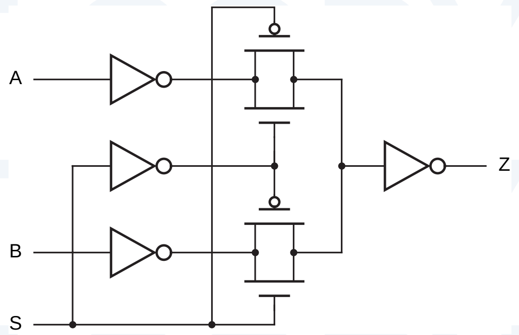


Figure X5.59

You should be able to design this circuit with no more than six SSI and MSI packages, but don't use more than 10 in any case.

5.56 Repeat Exercise 5.55 using ABEL. Does the design fit into a single GAL20V8?

5.57 Repeat Exercise 5.55 using VHDL.

5.58 Design a 3-input, 5-bit multiplexer that fits in a 24-pin IC package. Write the truth table and draw a logic diagram and logic symbol for your multiplexer.

5.59 Write the truth table and a logic diagram for the logic function performed by the CMOS circuit in Figure X5.59. (The circuit contains transmission gates, which were introduced in Section 3.7.1.)

5.60 A famous logic designer decided to quit teaching and make a fortune by fabricating huge quantities of the MSI circuit shown in Figure X5.64.

5-61 (a)Label the inputs and outputs of the circuit with appropriate signal names, including active-level indications.

5-62 (b)What does the circuit do? Be specific and account for all inputs and outputs.

5-63 (c)Draw the MSI logic symbol that would go on the data sheet of this wonderful device.

5-64 (d)With what standard MSI parts does the new part compete? Do you think it would be successful in the MSI marketplace?

barrel shifter

5.65 A 16-bit *barrel shifter* is a combinational logic circuit with 16 data inputs, 16 data outputs, and 4 control inputs. The output word equals the input word, rotated by a number of bit positions specified by the control inputs. For example, if the input word equals ABCDEFGHIJKLMNOP (each letter represents one bit), and the control inputs are 0101 (5), then the output word is FGHIJKLMNOPABCDE. Design a 16-bit barrel shifter using combinational MSI parts discussed in this chapter. Your design should contain 20 or fewer ICs. Do not draw a complete schematic, but sketch and describe your design in general terms and indicate the types and total number of ICs required.

5.66 Write an ABEL program for the barrel shifter in Exercise 5.65.

5.67 Write a VHDL program for the barrel shifter in Exercise 5.65.

5.68 Show how to realize the 4-input, 18-bit multiplexer with the functionality described in Table 5-39 using 18 74x151s.

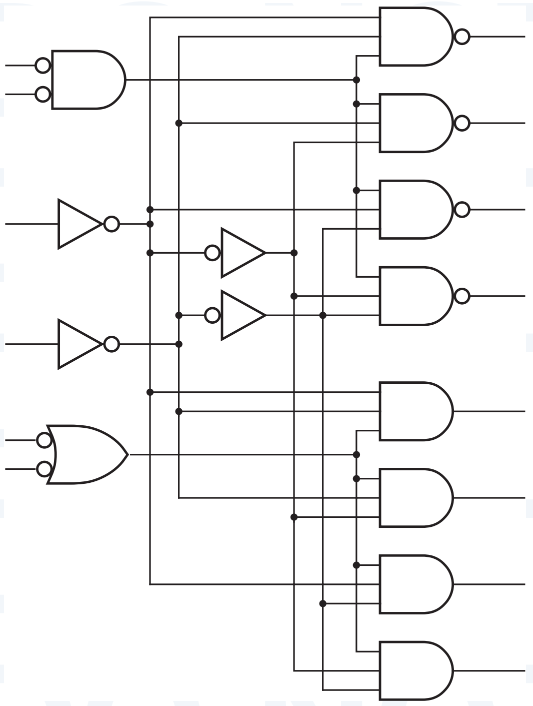


Figure X5.64

5.69 Show how to realize the 4-input, 18-bit multiplexer with the functionality of Table 5-39 using 9 74x153s and a “code converter” with inputs S2–S0 and outputs C1,C0 such that [C1,C0] = 00–11 when S2–S0 selects A–D, respectively.

5.70 Design a 3-input, 2-output combinational circuit that performs the code conversion specified in the previous exercise, using discrete gates.

5.71 Add a three-state-output control input `OE` to the VHDL multiplexer program in Table 5-42. Your solution should have only one process.

5.72 A digital designer who built the circuit in Figure 5-75 accidentally used 74x00s instead of ’08s in the circuit, and found that the circuit still worked, except for a change in the active level of the ERROR signal. How was this possible?

5.73 What logic function is performed by the CMOS circuit shown in Figure X5.73?

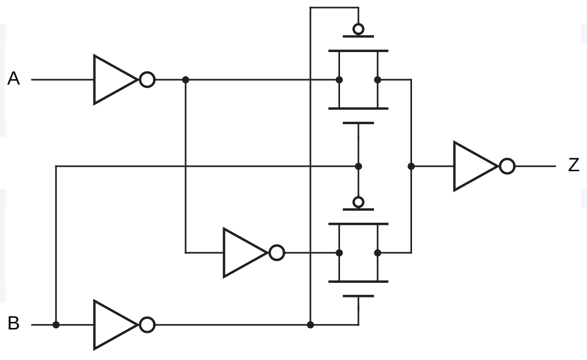


Figure X5.73

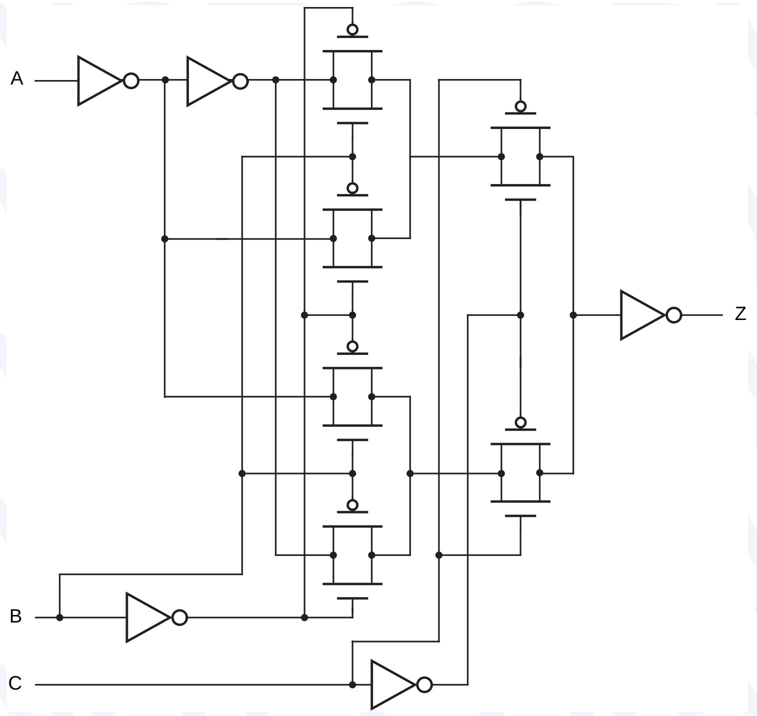


Figure X5.75

5.74 An odd-parity circuit with 2^n inputs can be built with 2^n-1 XOR gates. Describe two different structures for this circuit, one of which gives a minimum worst-case input to output propagation delay and the other of which gives a maximum. For each structure, state the worst-case number of XOR-gate delays, and describe a situation where that structure might be preferred over the other.

5.75 Write the truth table and a logic diagram for the logic function performed by the CMOS circuit in Figure X5.75.

5.76 Write a 4-step iterative algorithm corresponding to the iterative comparator circuit of Figure 5-79.

5.77 Design a 16-bit comparator using five 74x85s in a tree-like structure, such that the maximum delay for a comparison equals twice the delay of one 74x85.

5.78 Starting with a manufacturer's logic diagram for the 74x85, write a logic expression for the ALTBOUT output, and prove that it algebraically equals the expression derived in Drill 5.27.

5.79 Design a comparator similar to the 74x85 that uses the opposite cascading order. That is, to perform a 12-bit comparison, the cascading outputs of the high-order comparator would drive the cascading inputs of the mid-order comparator, and the mid-order outputs would drive the low-order inputs. You needn't do a complete logic design and schematic; a truth table and an application note showing the interconnection for a 12-bit comparison are sufficient.

5.80 Design a 24-bit comparator using three 74x682s and additional gates as required. Your circuit should compare two 24-bit unsigned numbers P and Q and produce two output bits that indicate whether $P = Q$ or $P > Q$.

5.81 Draw a 6-variable Karnaugh map for the s_2 function of Drill 5.29, and find all of its prime implicants. Using the 6-variable map format of Exercise 4.66, label the variables in the order x_0, y_0, x_2, y_2, x_1, y_1 instead of U, V, W, X, Y, Z. You need not write out the algebraic product corresponding to each prime implicant; simply identify each one with a number (1, 2, 3, …) on the map. Then make a list that shows for each prime implicant whether or not it is essential and how many inputs are needed on the corresponding AND gate.

5.82 Starting with the logic diagram for the 74x283 in Figure 5-90, write a logic expression for the S2 output in terms of the inputs, and prove that it does indeed equal the third sum bit in a binary addition as advertised. You may assume that $c_0 = 0$ (i.e., ignore c_0).

5.83 Using the information in Table 5-3, determine the maximum propagation delay from any A or B bus input to any F bus output of the 16-bit carry lookahead adder of Figure 5-95. You may use the “worst-case” analysis method.

5.84 Referring to the data sheet of a 74S182 carry lookahead circuit, determine whether or not its outputs match the equations given in Section 5.10.7.

5.85 Estimate the number of product terms in a minimal sum-of-products expression for the c_{32} output of a 32-bit binary adder. Be more specific than “billions and billions,” and justify your answer.

5.86 Draw the logic diagram for a 64-bit ALU using sixteen 74x181s and five 74S182s for full carry lookahead (two levels of ’182s). For the ’181s, you need only show the CIN inputs and G_L and P_L outputs.

5.87 Show how to build all four of the following functions using one SSI package and one 74x138.

$$F1 = X' \cdot Y' \cdot Z' + X \cdot Y \cdot Z \qquad F2 = X' \cdot Y' \cdot Z + X \cdot Y \cdot Z'$$

$$F3 = X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z \qquad F4 = X \cdot Y' \cdot Z' + X' \cdot Y \cdot Z$$

5.88 Design a customized decoder with the function table in Table X5.88 using MSI and SSI parts. Minimize the number of IC packages in your design.

Table X5.88

CS_L	A2	A1	A0	*Output to assert*
1	x	x	x	none
0	0	0	x	BILL_L
0	0	x	0	MARY_L
0	0	1	x	JOAN_L
0	0	x	1	PAUL_L
0	1	0	x	ANNA_L
0	1	x	0	FRED_L
0	1	1	x	DAVE_L
0	1	x	1	KATE_L

5.89 Repeat Exercise 5.88 using ABEL and a single GAL16V8.

5.90 Repeat Exercise 5.88 using VHDL.

5.91 Using ABEL and a single GAL16V8, design a customized multiplexer with four 3-bit input buses P, Q, R, T, and three select inputs S2–S0 that choose one of the buses to drive a 3-bit output bus Y according to Table X5.91.

Table X5.91

S2	S1	S0	*Input to select*
0	0	0	P
0	0	1	P
0	1	0	P
0	1	1	Q
1	0	0	P
1	0	1	P
1	1	0	R
1	1	1	T

5.92 Design a customized multiplexer with four 8-bit input buses P, Q, R, and T, selecting one of the buses to drive a 8-bit output bus Y according to Table X5.91. Use two 74x153s and a code converter that maps the eight possible values on S2–S0 to four select codes for the '153. Choose a code that minimizes the size and propagation delay of the code converter.

5.93 Design a customized multiplexer with five 4-bit input buses A, B, C, D, and E, selecting one of the buses to drive a 4-bit output bus T according to Table X5.93. You may use no more than three MSI and SSI ICs.

Table X5.93

S2	S1	S0	*Input to select*
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	E

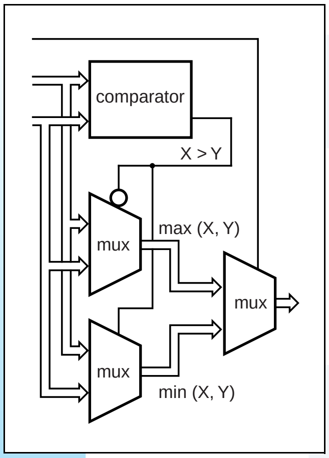
5.94 Repeat Exercise 5.93 using ABEL and one or more PAL/GAL devices from this chapter. Minimize the number and size of the GAL devices.

5.95 Design a 3-bit equality checker with six inputs, SLOT[2–0] and GRANT[2–0], and one active-low output, MATCH_L. The SLOT inputs are connected to fixed values when the circuit installed in the system, but the GRANT values are changed on a cycle-by-cycle basis during normal operation of the system. Using only SSI and MSI parts that appear in Tables 5-2 and 5-3, design a comparator with the shortest possible maximum propagation delay from GRANT[2–0] to MATCH_L. (*Note:*

The author had to solve this problem “in real life” to shave 2 ns off the critical-path delay in a 25-MHz system design.)

5.96 Design a combinational circuit whose inputs are two 8-bit unsigned binary integers, X and Y, and a control signal MIN/MAX. The output of the circuit is an 8-bit unsigned binary integer Z such that Z = 0 if X = Y; otherwise, Z = min(X,Y) if MIN/MAX = 1, and Z = max(X,Y) if MIN/MAX = 0.

5.97 Design a combinational circuit whose inputs are two 8-bit unsigned binary integers, X and Y, and whose output is an 8-bit unsigned binary integer Z = max(X,Y). For this exercise, you may use any of the 74x SSI and MSI components introduced in this chapter *except* the 74x682.



chapter 6

Combinational Design Examples

So far, we have looked at basic principles in several areas—number systems, digital circuits, and combinational logic—and we have described many of the basic building blocks of combinational design—decoders, multiplexers, and the like. All of that is a needed foundation, but the ultimate goal of studying digital design is eventually to be able to solve real problems by designing digital systems (well, duh…). That usually requires experience beyond what you can get by reading a textbook. We'll try to get you started by presenting a number of larger combinational design examples in this chapter.

The chapter is divided into three sections. The first section gives design examples using combinational building blocks. While the examples are written in terms of MSI functions, the same functions are widely used in ASIC and schematic-based FPGA design. The idea of these examples is to show that you can often express a combinational function using a collection of smaller building blocks. This is important for a couple of reasons: a hierarchical approach usually simplifies the overall design task, and the smaller building blocks often have a more efficient, optimized realization in FPGA and ASIC cells than what you'd get if you wrote a larger, monolithic description in an HDL and then just hit the "synthesize" button.

The second section gives design examples using ABEL. These designs are all targeted to small PLDs such as 16V8s and 20V8s. Besides the general use of the ABEL language, some of the examples illustrate the partitioning decisions that a designer must make when an entire circuit does not fit into a single component.

A VHDL-based approach is especially appropriate for larger designs that will be realized in a single CPLD, FPGA or ASIC, as described in the third section. You may notice that these examples do not target a specific CPLD or FPGA. Indeed, this is one of the benefits of HDL-based design; most or all of the design effort is “portable” and can be targeted to any of a variety of technologies.

The only prerequisites for this chapter are the chapters that precede it. The three sections are written to be pretty much independent of each other, so you don’t have to read about ABEL if you’re only interested in VHDL, or vice versa. Also, the rest of the book is written so that you can read this chapter now or skip it and come back later.

6.1 Building-Block Design Examples

6.1.1 Barrel Shifter

barrel shifter

A *barrel shifter* is a combinational logic circuit with *n* data inputs, *n* data outputs, and a set of control inputs that specify how to shift the data between input and output. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular, arithmetic, or logical), and the amount of shift (typically 0 to *n*–1 bits, but sometimes 1 to *n* bits).

In this subsection, we’ll look at the design of a simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input S[3:0] to specify the amount of shift. For example, if the input word is ABCDEFGHGIHKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHGIHKLMNOPABCDE.

From one point of view, this problem is deceptively simple. Each output bit can be obtained from a 16-input multiplexer controlled by the shift-control inputs, which each multiplexer data input connected to the appropriate On the other hand, when you look at the details of the design, you’ll see that there are trade-offs in the speed and size of the circuit.

Let us first consider a design that uses off-the-shelf MSI multiplexers. A 16-input, one-bit multiplexer can be built using two 74x151s, by applying S3 and its complement to the EN_L inputs and combining the Y_L data outputs with a NAND gate, as we showed in Figure 5-66 for a 32-input multiplexer. The low-order shift-control inputs, S2–S0. connect to the like-named select inputs of the ’151s.

We complete the design by replicating this 16-input multiplexer 16 times and hooking up the data inputs appropriately, as shown in Figure 6-1. The top ’151 of each pair is enabled by S3_L, and the bottom one by S3; the remaining select bits are connected to all 32 ’151s. Data inputs D0–D7 of each ’151 are connected to the DIN inputs in the listed order from left to right.

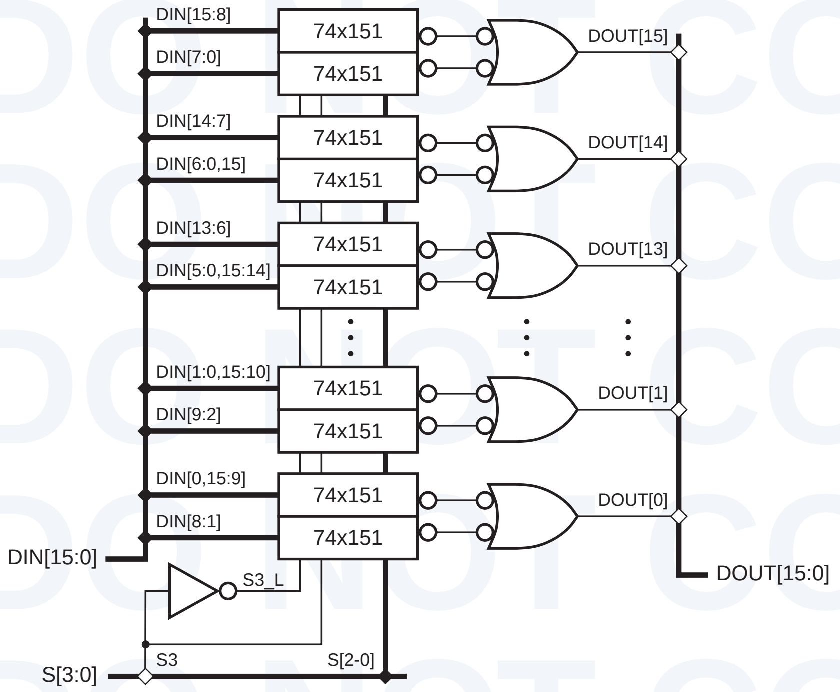


Figure 6-1
One approach to building a 16-bit barrel shifter.

The first row of Table 6-1 shows the characteristics of this first approach. About 36 chips (32 74x151s, 4 74x00s, and 1/6 74x04) are used in the MSI/SSI realization. We can reduce this to 32 chips by replacing the 74x151s with 74x251s and tying their three-state Y outputs together, as tabulated in the second row. Both of these designs have very heavy loading on the control inputs; each of the control bits S[2:0] must be connected to the like-named select input of all 32 multiplexers. The data inputs are also fairly heavily loaded; each data bit must connect to 16 different multiplexer data inputs, corresponding to the 16 possible shift amounts. However, assuming that the heavy control and data loads don't slow things down too much, the 74x251-based approach yields the shortest data delay, with each data bit passing through just one multiplexer.

Alternatively, we could build the barrel shifter using 16 74x157 2-input, 4-bit multiplexers, as tabulated in the last row of the table. We start by using four 74x157s to make a 2-input, 16-bit multiplexer. Then, we can hook up a first set

Multiplexer Component	*Data Loading*	*Data Delay*	*Control Loading*	*Total ICs*
74x151	16	2	32	36
74x251	16	1	32	32
74x153	4	2	8	16
74x157	2	4	4	16

Table 6-1
Properties of four different barrel-shifter design approaches.

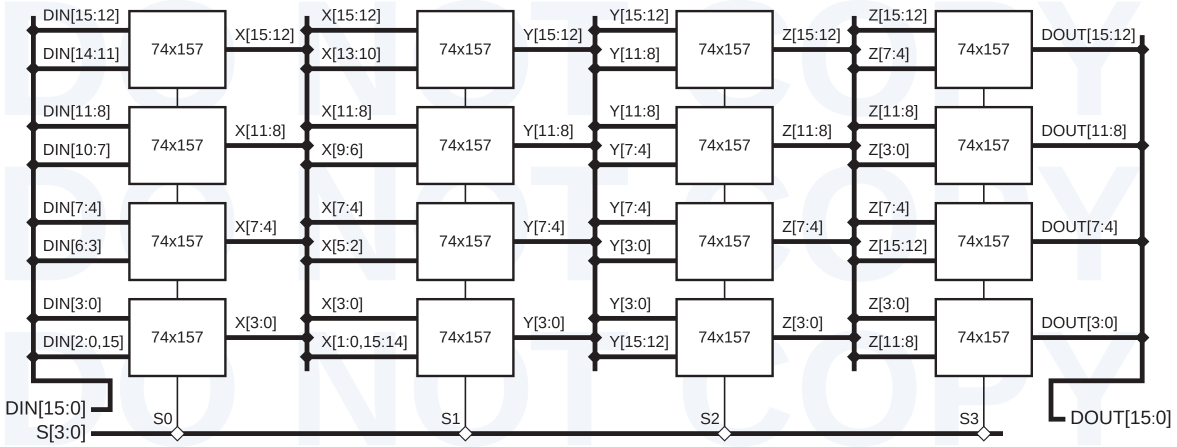


Figure 6-2 A second approach to building a 16-bit barrel shifter.

of four ’157s controlled by S0 to shift the input word left by 0 or 1 bit. The data outputs of this set are connected to the inputs of a second set, controlled by S1, which shifts its input word left by 0 or 2 bits. Continuing the cascade, a third and fourth set are controlled by S2 and S3 to shift selectively by 4 and 8 bits, as shown in Figure 6-2. Here, the 1A-4A and 1B-4B inputs and the 1Y-4Y outputs of each ’157 are connected to the indicated signals in the listed order from left to right.

The ’157-based approach requires only half as many MSI packages and has far less loading on the control and data inputs. On the other hand, it has the longest data-path delay, since each data bit must pass through four 74x157s.

Halfway between the two approaches, we can use eight 74x153 4-input, 2-bit multiplexers two build a 4-input, 16-bit multiplexer. Cascading two sets of these, we can use S[3:2] to shift selectively by 0, 4, 8, or 12 bits, and S[1:0] to shift by 0–3 bits. This approach has the performance characteristics shown in the third row of Table 6-1, and would appear to be the best compromise if you don’t need to have the absolutely shortest possible data delay.

The same kind of considerations would apply if you were building the barrel shifter out of ASIC cells instead of MSI parts, except you’d be counting chip area instead of MSI/SSI packages.

Typical ASIC cell libraries have 1-bit-wide multiplexers, usually realized with CMOS transmission gates, with 2 to 8 inputs. To build a larger multiplexer, you have to put together the appropriate combination of smaller cells. Besides the kind of choices we encountered in the MSI example, you have the further complication that CMOS delays are highly dependent on loading. Thus, depending on the approach, you must decide where to add buffers to the control lines, the data lines, or both to minimize loading-related delays. An approach that looks good on paper, before analyzing these delays and adding buffers, may actually turn out to have poorer delay or more chip area than another approach.

6.1.2 Simple Floating-Point Encoder

The previous example used multiple copies of a single building block, a multiplexer, and it was pretty obvious from the problem statement that a multiplexer was the appropriate building block. The next example shows that you sometimes have to look a little harder to see the solution in terms of known building blocks.

Now let's look at a design problem whose MSI solution is not quite so obvious, a "fixed-point to floating-point encoder." An unsigned binary integer B in the range $0 \le B < 2^{11}$ can be represented by 11 bits in "fixed-point" format, $B = b_{10}b_9 \ldots b_1b_0$. We can represent numbers in the same range with less precision using only 7 bits in a floating-point notation, $F = M \cdot 2^E$, where M is a 4-bit mantissa $m_3m_2m_1m_0$ and E is a 3-bit exponent $e_2e_1e_0$. The smallest integer in this format is $0 \cdot 2^0$ and the largest is $(2^4-1) \cdot 2^7$.

Given an 11-bit fixed-point integer B, we can convert it to our 7-bit floating-point notation by "picking off" four high-order bits beginning with the most significant 1, for example,

$$\begin{aligned}
11010110100 &= 1101 \cdot 2^7 + 0110100 \\
00100101111 &= 1001 \cdot 2^5 + 01111 \\
00000111110 &= 1111 \cdot 2^2 + 10 \\
00000001011 &= 1011 \cdot 2^0 + 0 \\
00000000010 &= 0010 \cdot 2^0 + 0
\end{aligned}$$

The last term in each equation is a truncation error that results from the loss of precision in the conversion. Corresponding to this conversion operation, we can write the specification for a fixed-point to floating-point encoder circuit:

- A combinational circuit is to convert an 11-bit unsigned binary integer B into a 7-bit floating-point number M,E, where M and E have 4 and 3 bits, respectively. The numbers have the relationship $B = M \cdot 2^E + T$, where T is the truncation error, $0 \le T < 2^E$.

Starting with a problem statement like the one above, it takes some creativity to come up with an efficient circuit design—the specification gives no clue. However, we can get some ideas by looking at how we converted numbers by hand earlier. We basically scanned each input number from left to right to find the first position containing a 1, stopping at the b_3 position if no 1 was found. We picked off four bits starting at that position to use as the mantissa, and the starting position number determined the exponent. These operations are beginning to sound like MSI building blocks.

"Scanning for the first 1" is what a generic priority encoder does. The output of the priority encoder is a number that tells us the position of the first 1. The position number determines the exponent; first-1 positions of $b_{10} - b_3$ imply exponents of 7–0, and positions of $b_2 - b_0$ or no-1-found imply an exponent of 0. Therefore, we can scan for the first 1 with an 8-input priority encoder with inputs

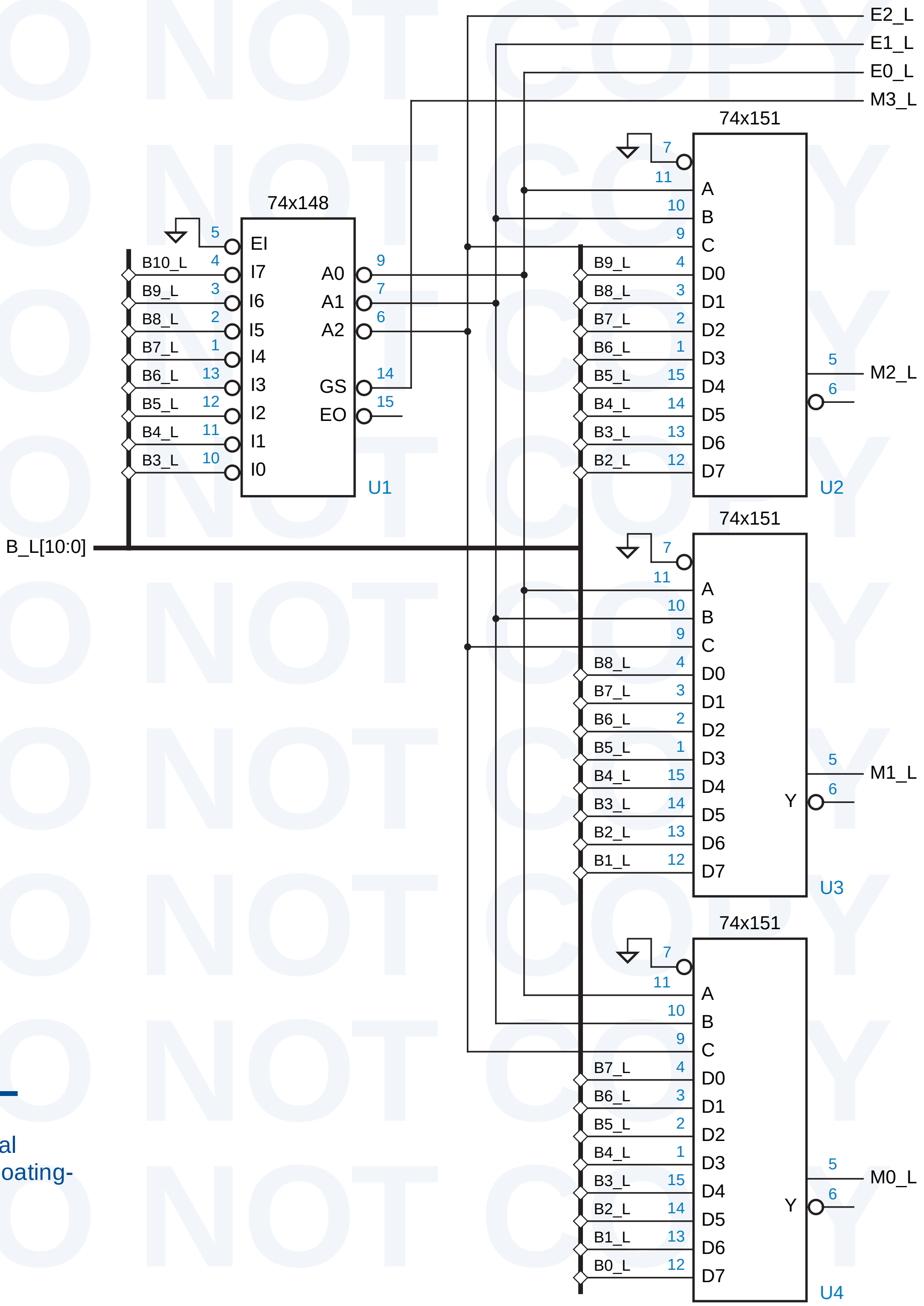


Figure 6-3
A combinational fixed-point to floating-point encoder.

I7 (highest priority) through I0 connected to $b_{10} - b_3$. We can use the priority encoder's A2 – A0 outputs directly as the exponent, as long as the no-1-found case produces A2 – A0 = 000.

"Picking off four bits" sounds like a "selecting" or multiplexing operation. The 3-bit exponent determines which four bits of *B* we pick off, so we can use the exponent bits to control an 8-input, 4-bit multiplexer that selects the appropriate four bits of *B* to form *M*.

An MSI circuit that results from these ideas is shown in Figure 6-3. It contains several optimizations:

- Since the available MSI priority encoder, the 74x148, has active-low inputs, the input number *B* is assumed to be available on an active-low bus B_L[10:0]. If only an active-high version of *B* is available, then eight inverters can be used to obtain the active-low version.
- If you think about the conversion operation a while, you'll realize that the most significant bit of the mantissa, m_3, is always 1, except in the no-1-found case. The '148 has a GS_L output that indicates this case, allowing us to eliminate the multiplexer for m_3.
- The '148 has active-low outputs, so the exponent bits (E0_L–E2_L) are produced in active-low form. Naturally, three inverters could be used to produce an active-high version.
- Since everything else is active-low, active-low mantissa bits are used too. Active-high bits are also readily available on the '148 EO_L and the '151 Y_L outputs.

Strictly speaking, the multiplexers in Figure 6-3 are drawn incorrectly. The 74x151 symbol can be drawn alternatively as shown in Figure 6-4. In words, if the multiplexer's data inputs are active low, then the data outputs have an active level opposite that shown in the original symbol. The "active-low-data" symbol

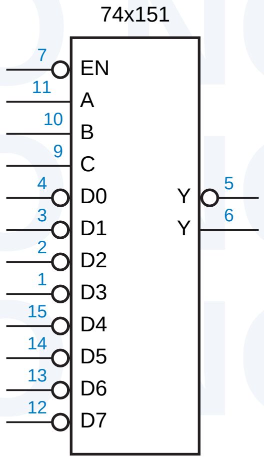


Figure 6-4
Alternate logic symbol for the 74x151 8-input multiplexer.

should be preferred in Figure 6-3, since the active levels of the ’151 inputs and outputs would then match their signal names. However, in data transfer and storage applications, designers (and the book) don’t always “go by the book.” It is usually clear from the context that a multiplexer (or a multibit register, in Section 8.2.5) does not alter the active level of its data.

6.1.3 Dual-Priority Encoder

Quite often MSI building blocks need a little help from their friends—ordinary gates— to get the job done. In this example, we’d like to build a priority encoder that identifies not only the highest but also the second-highest priority asserted signal among a set of eight request inputs.

We’ll assume for this example that the request inputs are active low and are named R0_L–R7_L, where R0_L has the highest priority. We’ll use A2–A0 and AVALID to identify the highest-priority request, where AVALID is asserted only if at least one request input is asserted. We’ll use B2–B0 and BVALID to identify the second-highest-priority request, where BVALID is asserted only if at least two request inputs are asserted.

Finding the highest-priority request is easy enough, we can just use a 74x148. To find the second highest-priority request, we can use another ’148, but only if we first “knock out” the highest-priority request before applying the request inputs. This can be done using a decoder to select a signal to knock out, based on A2–A0 and AVALID from the first ’148. These ideas are combined in the solution shown in Figure 6-6. A 74x138 decoder asserts at most one of its eight outputs, corresponding to the highest-priority request input. The outputs are fed to a rank of NAND gates to “turn off” the highest-priority request.

A trick is used in this solution is to get active-high outputs from the ’148s, as shown in Figure 6-5. We can rename the address outputs A2_L–A0_L to be active high if we also change the name of the request input that is associated with each output combination. In particular, we complement the bits of the request number. In the redrawn symbol, request input I0 has the highest priority.

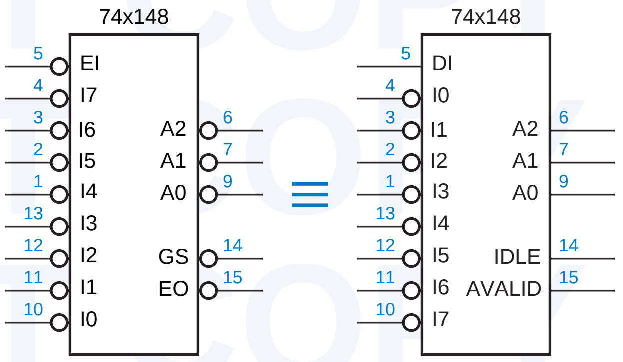


Figure 6-5
Alternate logic symbols for the 74x148 8-input priority encoder.

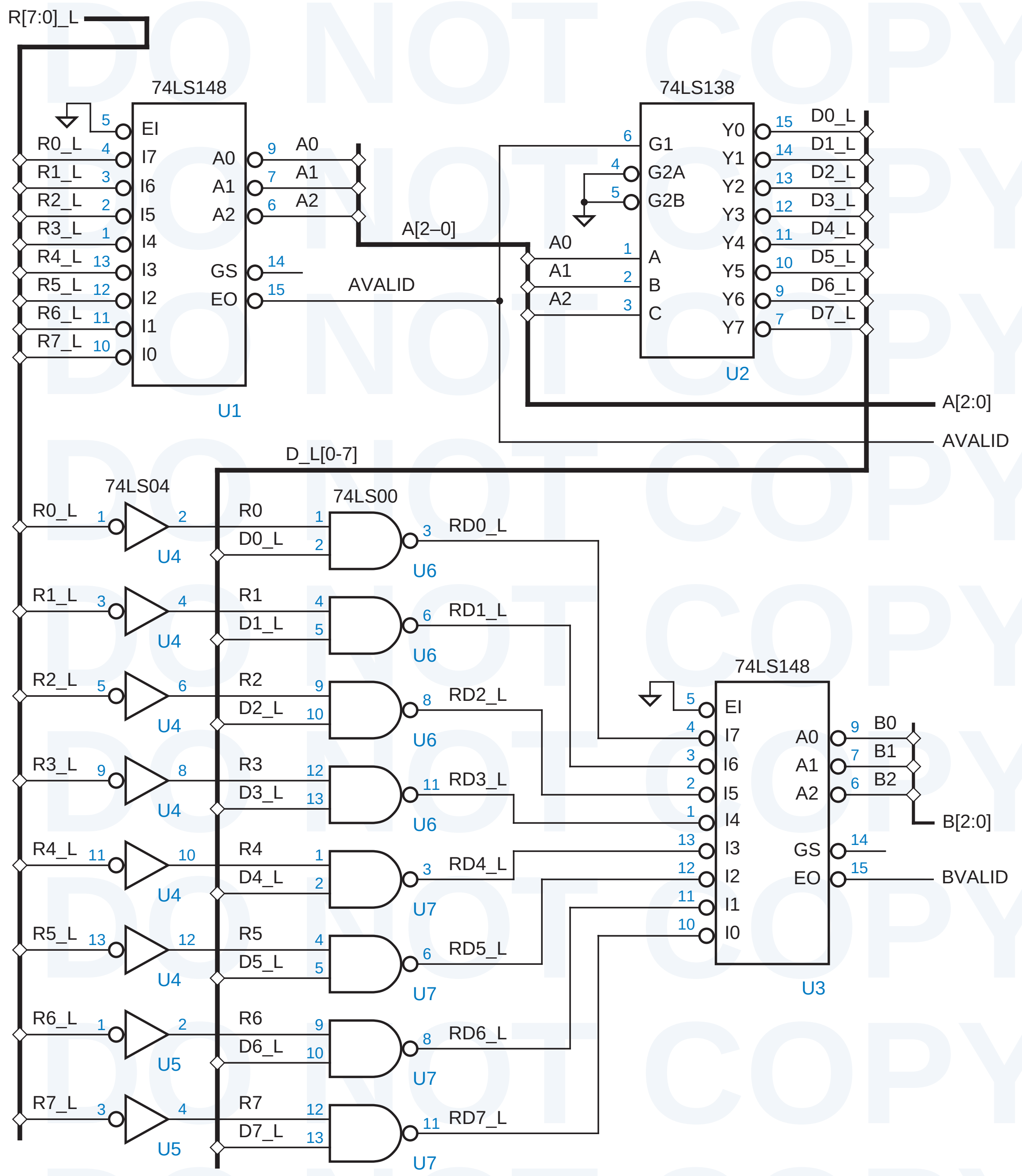


Figure 6-6 First-and second-highest priority encoder circuit.

6.1.4 Cascading Comparators

In Section 5.9.4, we showed how 74x85 4-bit comparators can be cascaded to create larger comparators. Since the 74x85 uses a serial cascading scheme, it can be used to build arbitrarily large comparators. The 74x682 8-bit comparator, on the other hand, doesn't have any cascading inputs and outputs at all. Thus, at first glance, you might think that it can't be used to build larger comparators. But that's not true.

If you think about the nature of a large comparison, it is clear that two wide inputs, say 32 bits (four bytes) each, are equal only if their corresponding bytes are equal. If we're trying to do a greater-than or less-than comparison, then the corresponding most-significant that are not equal determine the result of the comparison.

Using these ideas, Figure 6-7 uses three 74x682 8-bit comparators to do equality and greater-than comparison on two 24-bit operands. The 24-bit results are derived from the individual 8-bit results using combinational logic for the following equations:

$$\mathrm{PEQQ} = \mathrm{EQ2} \cdot \mathrm{EQ1} \cdot \mathrm{EQ0}$$
$$\mathrm{PGTQ} = \mathrm{GT2} + \mathrm{EQ2} \cdot \mathrm{GT1} + \mathrm{EQ2} \cdot \mathrm{EQ1} \cdot \mathrm{GT0}$$

This "parallel" expansion approach is actually faster than the 74x85's serial cascading scheme, because it does not suffer the delay of propagating the cascading signals through a cascade of comparators. The parallel approach can be used to build very wide comparators using two-level AND-OR logic to combine the 8-bit results, limited only by the fan-in constraints of the AND-OR logic. Arbitrary large comparators can be made if you use additional levels of logic to do the combining.

6.1.5 Mode-Dependent Comparator

Quite often, the requirements for a digital-circuit application are specified in a way that makes an MSI or other building-block solution obvious. For example, consider the following problem:

- Design a combinational circuit whose inputs are two 8-bit unsigned binary integers, X and Y, and a control signal MIN/MAX. The output of the circuit is an 8-bit unsigned binary integer Z such that Z = min(X,Y) if MIN/MAX = 1, and Z = max(X,Y) if MIN/MAX = 0.

This circuit is fairly easy to visualize in terms of MSI functions. Clearly, we can use a comparator to determine whether X > Y. We can use the comparator's output to control multiplexers that produce min(X,Y) and max(X,Y), and we can use another multiplexer to select one of these results depending on MIN/MAX. Figure 6-8(a) is the block diagram of a circuit corresponding to this approach.

Our first solution approach works, but it's more expensive than it needs to be. Although it has three two-input multiplexers, there are only two input words,

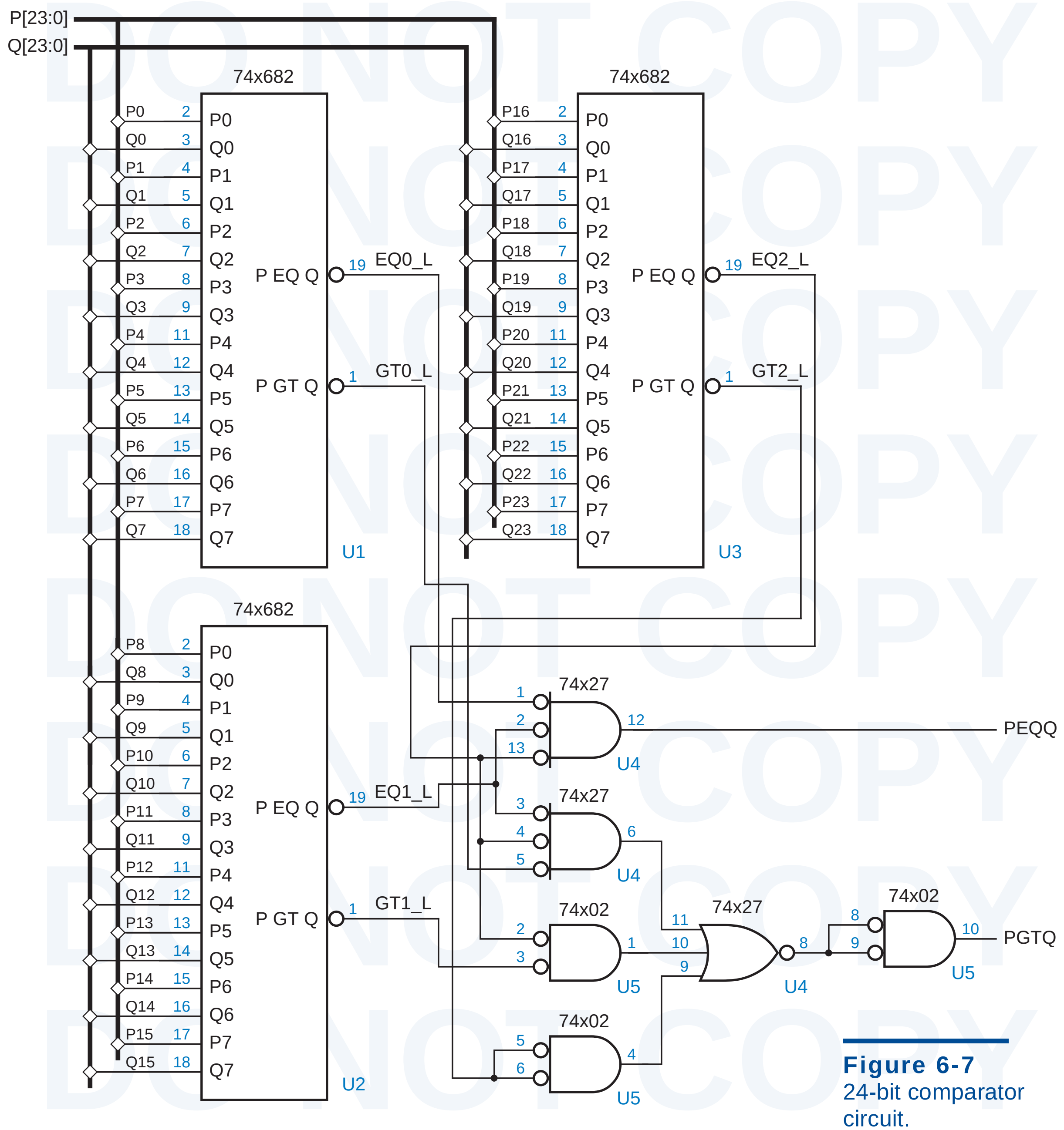


Figure 6-7
24-bit comparator circuit.

X and Y, that may ultimately be selected and produced at the output of the circuit. Therefore, we should be able to use just a single two-input mux, and use some other logic to figure out which input to tell it to select. This approach is shown in Figure 6-8(b) and (c). The “other logic” is very simple indeed, just a single XOR gate.

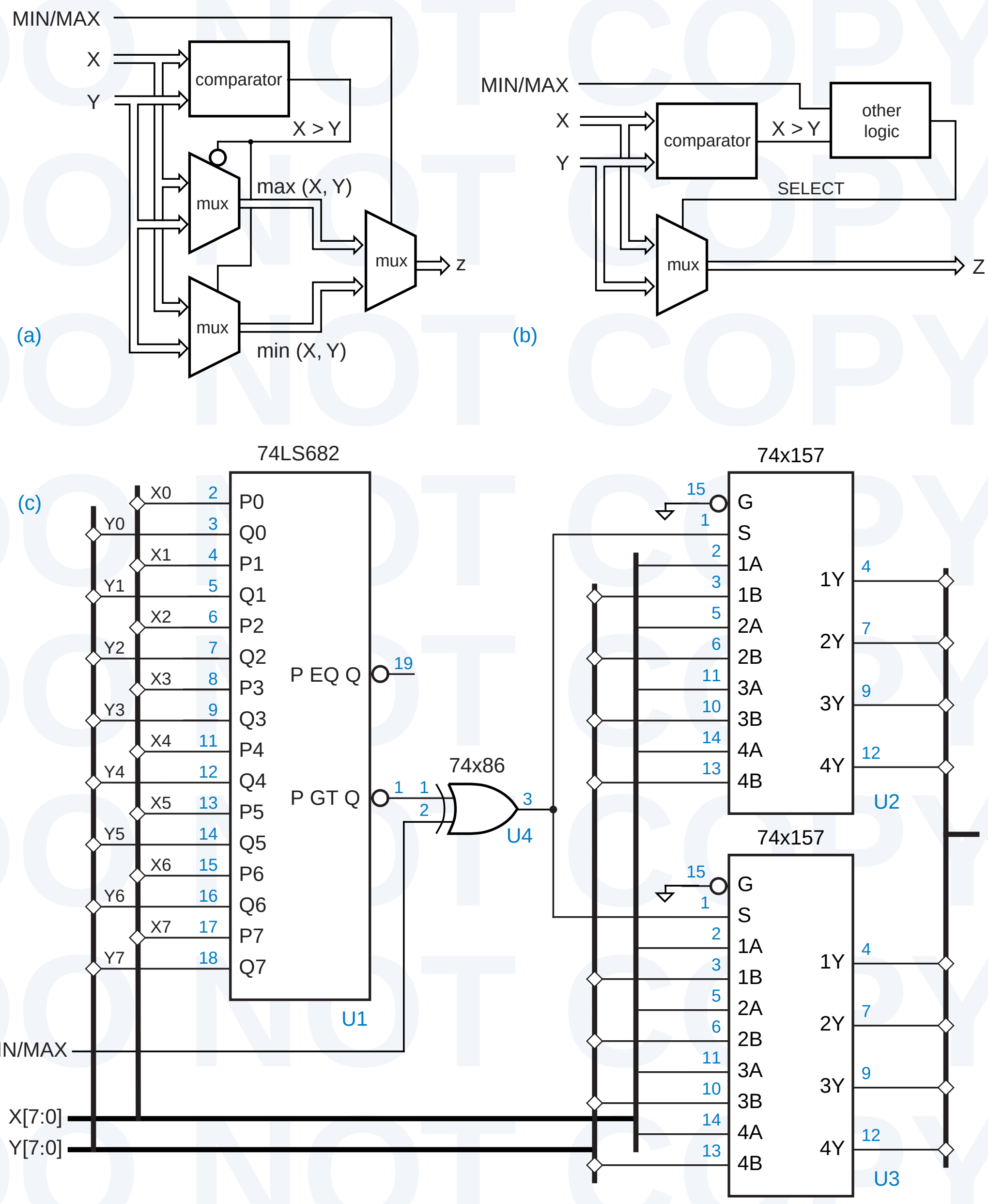


Figure 6-8 Mode-dependent comparator circuit: (a) block diagram of a "first-cut" solution; (b) block diagram of a more cost-effective solution; (c) logic diagram for the second solution.

DON'T BE A BLOCKHEAD

The wastefulness of our original design approach in Figure 6-8(a) may have been obvious to you from the beginning, but it demonstrates an important approach to designing with building blocks:

- Use standard building blocks to handle data, and look for ways that a single block can perform different functions at different times or in different modes. Design control circuits to select the appropriate functions as needed, to reduce the total parts count of the design.

As Figure 6-8(c) dramatically shows, this approach can save a lot of chips. When designing with IC chips, you should *not* heed the slogan, "Have all you want, we'll make more"!

6.2 Design Examples Using ABEL and PLDs

6.2.1 Barrel Shifter

A barrel shifter, defined on page 464, is good example of something *not* to design using PLDs. However, we can put ABEL to good use to describe a barrel shifter's function, and we'll also see why a typical barrel shifter is not a good fit for a PLD.

Table 6-2 shows the equations for a 16-bit barrel shifter with the same functionality as the example on page 464—it does left circular shifts only, using a 4-bit control input S[3..0] to specify the amount of shift. ABEL makes it easy to specify the functionality of the overall circuit without worrying about how the circuit might be partitioned into multiple chips. Also, ABEL dutifully generates a minimal sum-of-products expression for each output bit. In this case, each output requires 16 product terms.

Partitioning the 16-bit barrel shifter into multiple PLDs is a difficult task in two different ways. First, it should be obvious that the nature of the function is such that every output bit depends on every input bit. A PLD that produces, say, the DOUT0 output must have all 16 DIN inputs and all four S inputs available to it. So, a GAL16V8 definitely cannot be used; it has only 16 inputs.

The GAL20V8 is similar to the GAL16V8, with the addition of four input-only pins. If we use all 20 available inputs, we are left with two output-only pins (corresponding to the top and bottom outputs in Figure 5-27 on page 341). Thus, it seems possible that we could realize the barrel shifter using eight 20V8 chips, producing two output bits per chip.

No, we still have a problem. The second dimension of difficulty in a PLD-based barrel shifter is the number of product terms per output. The barrel shifter requires 16, and the 20V8 provides only 7. We're stuck—any realization of the barrel shifter in 20V8s is going to require multiple-pass logic. At this point, we would be best advised to look at partitioning options along the lines that we did in Section 6.1.1.

Table 6-2
ABEL program for a 16-bit barrel shifter.

```
module barrel16
title '16-bit Barrel Shifter'

" Inputs and Outputs
DIN15..DIN0, S3..S0                   pin;
DOUT15..DOUT0                         pin istype 'com';

S = [S3..S0];

equations

[DOUT15..DOUT0] = (S==0) & [DIN15..DIN0]
                # (S==1) & [DIN14..DIN0,DIN15]
                # (S==2) & [DIN13..DIN0,DIN15..DIN14]
                # (S==3) & [DIN12..DIN0,DIN15..DIN13]
                ...
                # (S==12) & [DIN3..DIN0,DIN15..DIN4]
                # (S==13) & [DIN2..DIN0,DIN15..DIN3]
                # (S==14) & [DIN1..DIN0,DIN15..DIN2]
                # (S==15) & [DIN0,DIN15..DIN1];

end barrel16
```

The 16-bit barrel shifter can be realized without much difficulty in a larger programmable device, that is, in a CPLD or an FPGA with enough I/O pins. However, imagine that we were trying to design a 32-bit or 64-bit barrel shifter. Clearly, we would need to use a device with even more I/O pins, but that's not all. The number of product terms and the large amount of connectivity (all the inputs connect to all the outputs) would still be challenging.

Indeed, a typical CPLD or FPGA fitter could have difficulty realizing a large barrel shifter with small delay or even at all. There is a critical resource that we took for granted in the partitioned, building-block barrel-shifter designs of Section 6.1.1—wires! An FPGA is somewhat limited in its internal connectivity, and a CPLD is even more so. Thus, even with modern FPGA and CPLD design tools, you may still have to "use your head" to partition the design in a way that helps the tools do their job.

Barrel shifters can be even more complex than what we've shown so far. Just for fun, Table 6-3 shows the design for a barrel shifter that supports six different kinds of shifting. This requires even more product terms, up to 40 per output! Although you'd never build this device in a PLD, CPLD, or small FPGA, the minimized ABEL equations are useful because they can help you understand the effects of some of your design choices. For example, by changing the coding of `SLA` and `SRA` to `[1,.X.,0]` and `[1,.X.,1]`, you can reduce the total number of product terms in the design from 624 to 608. You can save more product terms by changing the coding of the shift amount for some shifts (see Exercise 6.3). The savings from these changes may carry over to other design approaches.

Table 6-3 ABEL program for a multi-mode 16-bit barrel shifter.

```
module barrl16f
Title 'Multi-mode 16-bit Barrel Shifter'

" Inputs and Outputs
DIN15..DIN0, S3..S0, C2..C0         pin;
DOUT15..DOUT0                       pin istype 'com';

S = [S3..S0]; C = [C2..C0]; " Shift amount and mode
L = DIN15; R = DIN0;        " MSB and LSB

ROL = (C == [0,0,0]); " Rotate (circular shift) left
ROR = (C == [0,0,1]); " Rotate (circular shift) right
SLL = (C == [0,1,0]); " Shift logical left (shift in 0s)
SRL = (C == [0,1,1]); " Shift logical right (shift in 0s)
SLA = (C == [1,0,0]); " Shift left arithmetic (replicate LSB)
SRA = (C == [1,0,1]); " Shift right arithmetic (replicate MSB)

equations

[DOUT15..DOUT0] = ROL & (S==0) & [DIN15..DIN0]
                # ROL & (S==1) & [DIN14..DIN0,DIN15]
                # ROL & (S==2) & [DIN13..DIN0,DIN15..DIN14]
                ...
                # ROL & (S==15) & [DIN0,DIN15..DIN1]
                # ROR & (S==0) & [DIN15..DIN0]
                # ROR & (S==1) & [DIN0,DIN15..DIN1]
                ...
                # ROR & (S==14) & [DIN13..DIN0,DIN15..DIN14]
                # ROR & (S==15) & [DIN14..DIN0,DIN15]
                # SLL & (S==0) & [DIN15..DIN0]
                # SLL & (S==1) & [DIN14..DIN0,0]
                ...
                # SLL & (S==14) & [DIN1..DIN0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
                # SLL & (S==15) & [DIN0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
                # SRL & (S==0) & [DIN15..DIN0]
                # SRL & (S==1) & [0,DIN15..DIN1]
                ...
                # SRL & (S==14) & [0,0,0,0,0,0,0,0,0,0,0,0,0,0,DIN15..DIN14]
                # SRL & (S==15) & [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,DIN15]
                # SLA & (S==0) & [DIN15..DIN0]
                # SLA & (S==1) & [DIN14..DIN0,R]
                ...
                # SLA & (S==14) & [DIN1..DIN0,R,R,R,R,R,R,R,R,R,R,R,R,R,R]
                # SLA & (S==15) & [DIN0,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R]
                # SRA & (S==0) & [DIN15..DIN0]
                # SRA & (S==1) & [L,DIN15..DIN1]
                ...
                # SRA & (S==14) & [L,L,L,L,L,L,L,L,L,L,L,L,L,L,DIN15..DIN14]
                # SRA & (S==15) & [L,L,L,L,L,L,L,L,L,L,L,L,L,L,L,DIN15];
end barrl16f
```

6.2.2 Simple Floating-Point Encoder

We defined a simple floating-point number format on page 467, and posed the design problem of converting a number from fixed-point to this floating point format. The I/O-pin requirements of this design are limited—11 inputs and 7 outputs—so we can potentially use a single PLD to replace the four parts that were used in the MSI solution.

An ABEL program for the fixed-to-floating-point converter is given in Table 6-4. The `WHEN` statement expresses the operation of determining the exponent value `E` in a very natural way. Then `E` is used to select the appropriate bits of `B` to use as the mantissa `M`.

Despite the deep nesting of the `WHEN` statement, only four product terms are needed in the minimal sum for each bit of `E`. The equations for the M bits are not too bad either, requiring only eight product terms each. Unfortunately, the

Table 6-4 An ABEL program for the fixed-point to floating-point PLD.

```
module fpenc
title 'Fixed-point to Floating-point Encoder'
FPENC device 'P20L8';

" Input and output pins
B10..B0               pin 1..11;
E2..E0, M3..M0        pin 21..15 istype 'com';

" Constant expressions
B = [B10..B0];
E = [E2..E0];
M = [M3..M0];

equations

WHEN B < 16 THEN E = 0;
ELSE WHEN B < 32 THEN E = 1;
ELSE WHEN B < 64 THEN E = 2;
ELSE WHEN B < 128 THEN E = 3;
ELSE WHEN B < 256 THEN E = 4;
ELSE WHEN B < 512 THEN E = 5;
ELSE WHEN B < 1024 THEN E = 6;
ELSE E = 7;

M = (E==0) & [B3..B0]
  # (E==1) & [B4..B1]
  # (E==2) & [B5..B2]
  # (E==3) & [B6..B3]
  # (E==4) & [B7..B4]
  # (E==5) & [B8..B5]
  # (E==6) & [B9..B6]
  # (E==7) & [B10..B7];

end fpenc
```

GAL20V8 has available only seven product terms per output. However, the GAL22V10 (Figure 8-22 on page 684) has more product terms available, so we can use that if we like.

One drawback of the design in Table 6-4 is that the [M3..M0] outputs are slow; since they use [E2..E0], they take two passes through the PLD. A faster approach, if it fits, would be to rewrite the "select" terms (E==0, etc.) as intermediate equations before the equations section, and let ABEL expand the resulting M equations in a single level of logic. Unfortunately, ABEL does not allow WHEN statements outside of the equations section, so we'll have to roll up our sleeves and write our own logic expressions in the intermediate equations.

Table 6-5 shows the modified approach. The expressions for S7–S0 are just mutually-exclusive AND-terms that indicate exponent values of 7–0 depending on the location of the most significant 1 bit in the fixed-point input number. The exponent [E2..E0] is a binary encoding of the select terms, and the mantissa bits [M3..M0] are generated using a select term for each case. It turns out that these M equations still require 8 product terms per output bit, but at least they're a lot faster since they use just one level of logic.

Table 6-5 Alternative ABEL program for the fixed-point to floating-point PLD.

```
module fpence
title 'Fixed-point to Floating-point Encoder'
FPENCE device 'P20L8';

" Input and output pins
B10..B0                      pin 1..11;
E2..E0, M3..M0               pin 21..15 istype 'com';

" Intermediate equations
S7 = B10;
S6 = !B10 & B9;
S5 = !B10 & !B9 & B8;
S4 = !B10 & !B9 & !B8 & B7;
S3 = !B10 & !B9 & !B8 & !B7 & B6;
S2 = !B10 & !B9 & !B8 & !B7 & !B6 & B5;
S1 = !B10 & !B9 & !B8 & !B7 & !B6 & !B5 & B4;
S0 = !B10 & !B9 & !B8 & !B7 & !B6 & !B5 & !B4;

equations

E2 = S7 # S6 # S5 # S4;
E1 = S7 # S6 # S3 # S2;
E0 = S7 # S5 # S3 # S1;

[M3..M0] = S0 & [B3..B0] # S1 & [B4..B1] # S2 & [B5..B2]
         # S3 & [B6..B3] # S4 & [B7..B4] # S5 & [B8..B5]
         # S6 & [B9..B6] # S7 & [B10..B7];

end fpenc
```

6.2.3 Dual-Priority Encoder

In this example, we'll design a PLD-based priority encoder that identifies both the highest-priority and the second-highest-priority asserted signal among a set of eight active-high request inputs named [R0..R7], where R0 has the highest priority. We'll use [A2..A0] and AVALID to identify the highest-priority request, asserting AVALID only if a highest-priority request is present. Similarly, we'll use [B2:B0] and BVALID to identify the second-highest-priority request.

Table 6-6 shows an ABEL program for the priority encoder. As usual, a nested WHEN statement is perfect for expressing priority behavior. To find the

Table 6-6 ABEL program for a dual priority encoder.

```
title 'Dual Priority Encoder'
PRIORTWO device 'P16V8';

" Input and output pins
R7..R0                            pin 1..8;
AVALID, A2..A0, BVALID, B2..B0    pin 19..12 istype 'com';

" Set definitions
A = [A2..A0]; B = [B2..B0];

equations

WHEN R0==1 THEN A=0;
ELSE WHEN R1==1 THEN A=1;
ELSE WHEN R2==1 THEN A=2;
ELSE WHEN R3==1 THEN A=3;
ELSE WHEN R4==1 THEN A=4;
ELSE WHEN R5==1 THEN A=5;
ELSE WHEN R6==1 THEN A=6;
ELSE WHEN R7==1 THEN A=7;

AVALID = ([R7..R0] != 0);

WHEN (R0==1) & (A!=0) THEN B=0;
ELSE WHEN (R1==1) & (A!=1) THEN B=1;
ELSE WHEN (R2==1) & (A!=2) THEN B=2;
ELSE WHEN (R3==1) & (A!=3) THEN B=3;
ELSE WHEN (R4==1) & (A!=4) THEN B=4;
ELSE WHEN (R5==1) & (A!=5) THEN B=5;
ELSE WHEN (R6==1) & (A!=6) THEN B=6;
ELSE WHEN (R7==1) & (A!=7) THEN B=7;

BVALID = (R0==1) & (A!=0) # (R1==1) & (A!=1)
       # (R2==1) & (A!=2) # (R3==1) & (A!=3)
       # (R4==1) & (A!=4) # (R5==1) & (A!=5)
       # (R6==1) & (A!=6) # (R7==1) & (A!=7);

end priortwo
```

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  ----------
   8/1        8       1     Pin   AVALID
   4/5        8       1     Pin   A2
   4/5        8       1     Pin   A1
   4/5        8       1     Pin   A0
  24/8       11       1     Pin   BVALID
  24/17      11       1     Pin   B2
  20/21      11       1     Pin   B1
  18/22      11       1     Pin   B0
=========
 106/84          Best P-Term Total: 76
                        Total Pins: 16
                Average P-Term/Output: 9
```

Table 6-7 Product-term usage in the dual priority encoder PLD.

second-highest priority input, we exclude an input if its input number matches the highest-priority input number, which is `A`. Thus, we're using two-pass logic to compute the `B` outputs. The equation for `AVALID` is easy; `AVALID` is 1 if the request inputs are not all 0. To compute `BVALID`, we OR all of the conditions that set `B` in the `WHEN` statement.

Even with two-pass logic, the `B` outputs use too many product terms to fit in a 16V8; Table 6-7 shows the product-term usage. The `B` outputs use too many terms even for a 22V10, which has 16 terms for two of its output pins, and 8–14 for the others. Sometimes you just have to work harder to make things fit!

So, how can we save some product terms? One important thing to notice is that R0 can never be the second-highest priority asserted input, and therefore `B` can never be valid and 0. Thus, we can eliminate the `WHEN` clause for the `R0==1` case. Making this change reduces the minimum number of terms for `B2`–`B0` to 14, 17, and 15, respectively. We can almost fit the design in a 22V10, if we can just know one term out of the `B1` equation.

Well let's try something else. The second `WHEN` clause, for the `R0==2` case, also fails to make use of everything we know. We don't need the full generality

SUMS OF PRODUCTS AND PRODUCTS OF SUMS (SAY THAT 5 TIMES FAST)

You may recall from Section 4.3.6 that the minimal sum-of-products expression for the complement of a function can be manipulated through DeMorgan's theorem to obtain a minimal product-of-sums expression for the original function. You may also recall that the number of product terms in the minimal sum of products may differ from the number of sum terms in the minimal product of sums. The "P-terms" column in Table 6-7 lists the number of terms in both minimal forms (product/sum terms). If *either* minimal form has less than or equal to the number of product terms available in a 22V10's AND-OR array, then the function can be made to fit.

of `A!=0`; this case is only important when `R0` is 1. So, let us replace the first two lines of the original `WHEN` statement with

```
WHEN (R1==1) & (R0==1) THEN B=1;
```

This subtle change reduces the minimum number of terms for `B2`–`B0` to 12, 16, and 13, respectively. We made it! Can the number of product terms be reduced further, enough to fit into a 16V8 while maintaining the same functionality? It's not likely, but we'll leave that as an exercise (6.4) for the reader!

6.2.4 Cascading Comparators

We showed in Section 5.9.5 that equality comparisons are easy to realize in PLDs, but that magnitude comparisons (greater-than or less-than) of more than a few bits are not good candidates for PLD realization due to the large number of product terms required. Thus, comparators are best realized using discrete MSI comparator components or as specialized cells within an FPGA or ASIC library. However, PLDs are quite suitable for realizing the combinational logic used in "parallel expansion" schemes that construct wider comparators from smaller ones, as we'll show here.

In Section 5.9.4, we showed how to connect 74x85 4-bit comparators in series to create larger comparators. Although a serial cascading scheme requires no extra logic to build arbitrarily large comparators, it has the major drawback that the delay increases linearly with the length of the cascade.

In Section 6.1.4, on the other hand, we showed how multiple copies of the 74x682 8-bit comparator could be used in parallel along with combinational logic to perform a 24-bit comparison. This scheme can be generalized for comparisons of arbitrary width.

Table 6-8 is an ABEL program that uses a GAL22V10 to perform a 64-bit comparison using eight 74x682s to combine the equal (EQ) and greater-than (GT) outputs from the individual byte to produce all six possible relations of the two 64-bit input values (=, ≠, >, ≥, <, ≤).

In this program, the `PEQQ` and `PNEQ` outputs can be realized with one product term each. The remaining eight outputs use eight product terms each. As we've mentioned previously, the 22V10 provides 8-16 product terms per output, so the design fits.

HAVE IT YOUR WAY

Early PLDs such as the PAL16L8s did not have output-polarity control. Designers who used these devices were forced to choose a particular polarity, active high or active low, for some outputs in order to obtain reduced equations that would fit. When a 16V8, 20V8, 22V10, or any of a plethora of modern CPLDs is used, no such restriction exists. If an equation *or its complement* can be reduced to the number of product terms available, then the corresponding output can be made active high or active low by programming the output-polarity fuse appropriately.

Table 6-8
ABEL program for combining eight 74x682s into a 64-bit comparator.

```
module compexp
title 'Expansion logic for 64-bit comparator'
COMPEXP device 'P22V10';

" Inputs from the individual comparators, active-low, 7 = MSByte
EQ_L7..EQ_L0, GT_L7..GT_L0               pin 1..11, 13..14, 21..23;

" Comparison outputs
PEQQ, PNEQ, PGTQ, PGEQ, PLTQ, PLEQ       pin 15..20 istype 'com';

" Active-level conversions
EQ7 = !EQ_L7; EQ6 = !EQ_L6; EQ5 = !EQ_L5; EQ4 = !EQ_L4;
EQ3 = !EQ_L3; EQ2 = !EQ_L2; EQ1 = !EQ_L1; EQ0 = !EQ_L0;
GT7 = !GT_L7; GT6 = !GT_L6; GT5 = !GT_L5; GT4 = !GT_L4;
GT3 = !GT_L3; GT2 = !GT_L2; GT1 = !GT_L1; GT0 = !GT_L0;

" Less-than terms
LT7 = !(EQ7 # GT7); LT6 = !(EQ6 # GT6); LT5 = !(EQ5 # GT5);
LT4 = !(EQ4 # GT4); LT3 = !(EQ3 # GT3); LT2 = !(EQ2 # GT2);
LT1 = !(EQ1 # GT1); LT0 = !(EQ0 # GT0);

equations

PEQQ = EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & EQ0;

PNEQ = !(EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & EQ0);

PGTQ = GT7  #  EQ7 & GT6  #  EQ7 & EQ6 & GT5
     # EQ7 & EQ6 & EQ5 & GT4  #  EQ7 & EQ6 & EQ5 & EQ4 & GT3
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & GT2
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & GT1
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & GT0;

PLEQ = !(GT7  #  EQ7 & GT6  #  EQ7 & EQ6 & GT5
       # EQ7 & EQ6 & EQ5 & GT4  #  EQ7 & EQ6 & EQ5 & EQ4 & GT3
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & GT2
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & GT1
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & GT0);

PLTQ = LT7  #  EQ7 & LT6  #  EQ7 & EQ6 & LT5
     # EQ7 & EQ6 & EQ5 & LT4  #  EQ7 & EQ6 & EQ5 & EQ4 & LT3
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & LT2
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & LT1
     # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & LT0;

PGEQ = !(LT7  #  EQ7 & LT6  #  EQ7 & EQ6 & LT5
       # EQ7 & EQ6 & EQ5 & LT4  #  EQ7 & EQ6 & EQ5 & EQ4 & LT3
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & LT2
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & LT1
       # EQ7 & EQ6 & EQ5 & EQ4 & EQ3 & EQ2 & EQ1 & LT0);

end compexp
```

Table 6-9 Mode-control bits for the mode-dependent comparator.

M1	*M0*	*Comparison*
0	0	32-bit
0	1	31-bit
1	0	30-bit
1	1	not used

6.2.5 Mode-Dependent Comparator

For the next example, let us suppose we have a system in which we need to compare two 32-bit words under normal circumstances, but where we must sometimes ignore one or two low-order bits of the input words. The operating mode is specified by two mode-control bits, M1 and M0, as shown in Table 6-9.

As we've noted previously, comparing, adding, and other "iterative" operations are usually poor candidates for PLD-based design, because an equivalent two-level sum-of-products expression has far too many product terms. In Section 5.9.5, we calculated how many product terms are needed for an *n*-bit comparator. Based on these results, we certainly wouldn't be able to build the 32-bit mode-dependent comparator or even an 8-bit slice of it with a PLD; the 74x682 8-bit comparator is just about the most efficient possible single chip we can use to perform an 8-bit comparison. However, a PLD-based design is quite reasonable for handling the mode-control logic and the part of the comparison that is dependent on mode (the two low-order bits).

Figure 6-9 shows a complete circuit design resulting from this idea, and Table 6-11 is the ABEL program for a 16V8 MODECOMP PLD that handles the "random logic." Four '682s are used to compare most of the bits, and the 16V8 combines the '682 outputs and handles the two low-order bits as a function of the mode. Intermediate expressions `EQ30` and `GT30` are defined to save typing in the `equations` section of the program.

As shown in Table 6-10, the `XEQY` and `XGTY` outputs use 7 and 11 product terms, respectively. Thus, `XGTY` does not fit into the 7 product terms available on a 16V8 output. However, this is another example where we have some flexibility in our coding choices. By changing the coding of `MODE30` to `[1,.X.]`, we can reduce the product-term requirements for `XGTY` to 7/12, and thereby fit the design into a 16V8.

Table 6-10 Product-term usage for the MODECOMP PLD.

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  --------
   7/9       10        1    Pin   XEQY
  11/13      14        1    Pin   XGTY
=========
  18/22          Best P-Term Total: 18
                        Total Pins: 16
              Average P-Term/Output: 9
```

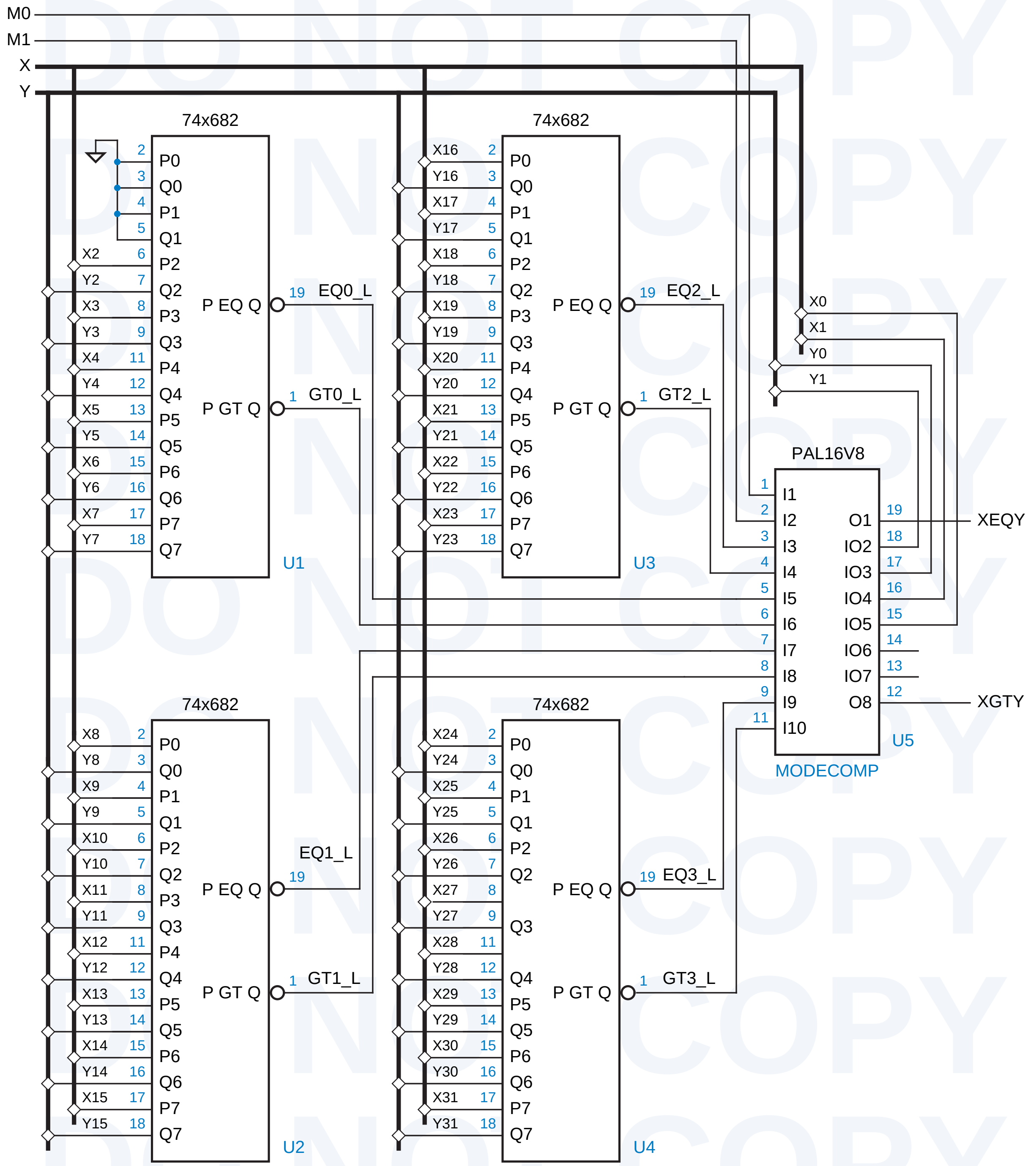


Figure 6-9 A 32-bit mode-dependent comparator.

Table 6-11
ABEL program for combining eight 74x682s into a 64-bit comparator.

```
module modecomp
title 'Control PLD for Mode-Dependent Comparator'
MODECOMP device 'P16V8';

" Input and output pins
M0, M1, EQ2_L, GT2_L, EQ0_L, GT0_L           pin 1..6;
EQ1_L, GT1_L, EQ3_L, GT3_L, X0, X1, Y0, Y1   pin 7..9, 10, 15..18;
XEQY, XGTY                                  pin 19, 12 istype 'com';

" Active-level conversions
EQ3 = !EQ3_L; EQ2 = !EQ2_L; EQ1 = !EQ1_L; EQ0 = !EQ0_L;
GT3 = !GT3_L; GT2 = !GT2_L; GT1 = !GT1_L; GT0 = !GT0_L;

" Mode definitions
MODE32 = ([M1,M0] == [0,0]); " 32-bit comparison
MODE31 = ([M1,M0] == [0,1]); " 31-bit comparison
MODE30 = ([M1,M0] == [1,0]); " 30-bit comparison
MODEXX = ([M1,M0] == [1,1]); " Unused

" Expressions for 30-bit equal and greater-than
EQ30 = EQ3 & EQ2 & EQ1 & EQ0;
GT30 = GT3 # (EQ3 & GT2) # (EQ3 & EQ2 & GT1) # (EQ3 & EQ2 & EQ1 & GT0);

equations

WHEN MODE32 THEN {
  XEQY = EQ30 & (X1==Y1) & (X0==Y0);
  XGTY = GT30 # (EQ30 & (X1>Y1)) # (EQ30 & (X1==Y1) & (X0>Y0));
  }
ELSE WHEN MODE31 THEN {
  XEQY = EQ30 & (X1==Y1);
  XGTY = GT30 # (EQ30 & (X1>Y1));
  }
ELSE WHEN MODE30 THEN {
  XEQY = EQ30;
  XGTY = GT30;
  }

end modecomp
```

6.2.6 Ones Counter

There are several important algorithms that include the step of counting the number of “1” bits in a data word. In fact, some microprocessor instruction sets have been extended recently to include ones counting as a basic instruction.

Counting the ones in a data word can be done easily as an iterative process, where you scan the word from one end to the other and increment a counter each time a “1” is encountered. However, this operation must be done more quickly inside the arithmetic and logic unit of a microprocessor. Ideally, we would like

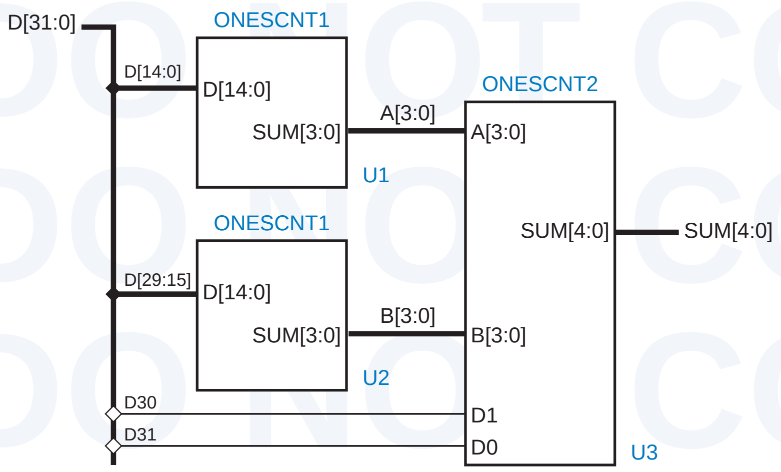


Figure 6-10 Possible partitioning for the ones-counting circuit.

ones counting to run as fast as any other arithmetic operation, such as adding two words. Therefore, a combinational circuit is required.

In this example, let us suppose that we have a requirement to build a 32-bit ones counter as part of a larger system. Based on the number of inputs and outputs required, we obviously can't fit the design into a single 22V10-class PLD, but we might be able to partition the design into a reasonably small number of PLDs.

Figure 6-10 shows such a partition. Two copies of a first 22V10, ONESCNT1, are used to count the ones in two 15-bit chunks of the 32-bit input word D[31:0], each producing a 4-bit sum output. A second 22V10, ONESCNT2, is used to add the two four bit sums and the last two input bits.

The program for ONESCNT1 is deceptively simple, as shown in Table 6-12. The statement "@CARRY 1" is included to limit the carry chain to one stage; as explained in Section 5.10.8, this reduces product-term requirements at the expense of helper outputs and increased delay.

Table 6-12 ABEL program for counting the 1 bits in a 15-bit word.

```
module onescnt1
title 'Count the ones in a 15-bit word'
ONESCNT1 device 'P22V10';

" Input and output pins
D14..D0             pin 1..11, 13..15, 23;
SUM3..SUM0          pin 17..20 istype 'com';

equations

@CARRY 1;
[SUM3..SUM0] = D0 + D1 + D2 + D3 + D4 + D5 + D6 + D7
             + D8 + D9 + D10 + D11 + D12 + D13 + D14;

end onescnt1
```

Unfortunately, when I compiled this program, my computer just sat there, CPU-bound, for an hour without producing any results. That gave me time to use my brain, a good exercise for those of us who have become too dependent on CAD tools. I then realized that I could write the logic function for the SUM0 output by hand in just a few seconds,

$$\text{SUM0} = \text{D0} \oplus \text{D1} \oplus \text{D2} \oplus \text{D3} \oplus \text{D4} \oplus \text{D5} \oplus \text{D6} \oplus \text{D7} \oplus \ldots \oplus \text{D13} \oplus \text{D14}$$

The Karnaugh map for this function is a checkerboard, and the minimal sum-of-products expression has 2^{14} product terms. Obviously this is not going to fit in one or a few passes through a 22V10! So, anyway, I killed the ABEL compiler process, and rebooted Windows just in case the compiler had gone awry.

Obviously, a partitioning into smaller chunks is required to design the ones-counting circuit. Although we could pursue this further using ABEL and PLDs, it's more interesting to do a structural design using VHDL, as we will in Section 6.3.6. The ABEL and PLD version is left as an exercise (6.6).

6.2.7 Tic-Tac-Toe

In this example, we'll design a combinational circuit that picks a player's next move in the game of Tic-Tac-Toe. The first thing we'll do is decide on a strategy for picking the next move. Let us try to emulate the typical human's strategy, by following the decision steps below:

1. Look for a row, column, or diagonal that has two of my marks (X or O, depending on which player I am) and one empty cell. If one exists, place my mark in the empty cell; I win!
2. Else, look for a row, column, or diagonal that has two of my opponent's marks and one empty cell. If one exists, place my mark in the empty cell to block a potential win by my opponent.
3. Else, pick a cell based on experience. For example, if the middle cell is open, it's usually a good bet to take it. Otherwise, the corner cells are good bets. Intelligent players can also notice and block a developing pattern by the opponent or "look ahead" to pick a good move.

TIC-TAC-TOE, IN CASE YOU DIDN'T KNOW

The game of Tic-Tac-Toe is played by two players on a 3 × 3 grid of cells that are initially empty. One player is "X" and the other is "O". The players alternate in placing their mark in an empty cell; "X" always goes first. The first player to get three of his or her own marks in the same row, column, or diagonal wins. Although the first player to move (X) has a slight advantage, it can be shown that a game between two intelligent players will always end in a draw; neither player will get three in a row before the grid fills up.

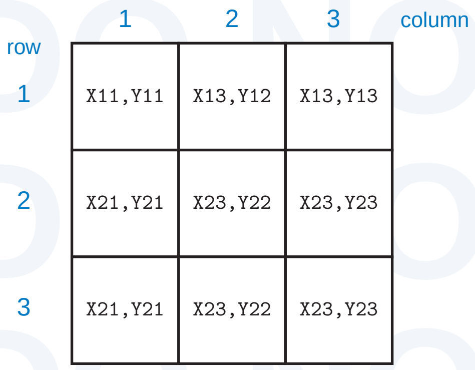


Figure 6-11
Tic-Tac-Toe grid and ABEL signal names.

Planning ahead, we'll call the second player "Y" to avoid confusion between "O" and "0" in our programs. The next thing to think about is how we might encode the inputs and outputs of the circuit. There are only nine possible moves that a player can make, so the output can be encoded in just four bits. The circuit's input is the current state of the playing grid. There are nine cells, and each cell has one of three possible states (empty, occupied by X, occupied by Y).

There are several choices of how to code the state of one cell. Because the game is symmetric, we'll choose a symmetric encoding that may help us later:

00	Cell is empty.
10	Cell is occupied by X.
01	Cell is occupied by Y.

So, we can encode the 3×3 grid's state in 18 bits. As shown in Figure 6-11, we'll number the grid with row and column numbers, and use ABEL signals `Xij` and `Yij` to denote the presence of X or Y in cell `i,j`. We'll look at the output coding later.

With a total of 18 inputs and 4 outputs, the Tic-Tac-Toe circuit could conceivably fit in just one 22V10. However, experience suggests that there's just no way. We're going to have to find a partitioning of the function, and partitioning along the lines of the decision steps on the preceding page seems like a good idea.

In fact, steps 1 and 2 are very similar; they differ only in reversing the roles of the player and the opponent. Here's where our symmetric encoding can pay

COMPACT ENCODING

Since each cell in the Tic-Tac-Toe grid can have only three states, not four, the total number of board configurations is 3^9, or 19,683. This is less than 2^{15}, so the board state can be encoded in only 15 bits. However, such an encoding would lead to much larger circuits for picking a move, unless the move-picking circuit was a read-only memory (see Exercise 11.26).

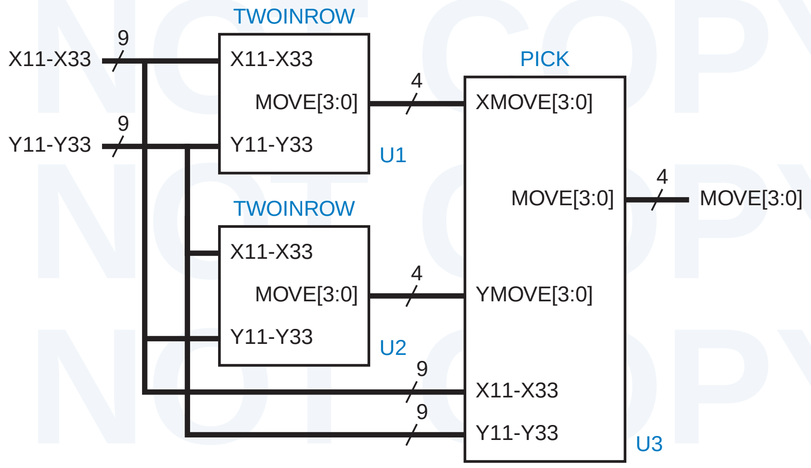


Figure 6-12
Preliminary PLD partitioning for the Tic-Tac-Toe game.

off. A PLD that finds me two of my marks in a row along with one empty cell for a winning move (step 1) can find two of my opponent's marks in a row plus an empty for a blocking move (step 2). All we have to do is swap the encodings for X and Y. With out selected coding, that doesn't require any logic, just physically swapping the `Xij` and `Yij` signals for each cell. With this in mind, we can use two copies of the same PLD, TWOINROW, to perform steps 1 and 2 as shown in Figure 6-12. Notice that the `X11`–`X33` signals are connected to the top inputs of the first TWOINROW PLD, but to the bottom inputs of the second.

The moves from the two TWOINROW PLDs can be examined in another PLD, PICK. This device picks a move from the first two PLDs if either found one; else it performs step 3. It looks like PICK has too many inputs and outputs to fit in a 22V10, but we'll come back to that later.

Table 6-13 is a program for the TWOINROW PLD. It looks at the grid's state from the point of view of X, that is, it looks for a move where X can get three in a row. The program makes extensive use of intermediate equations to define

Table 6-13
ABEL program to find two in a row in Tic-Tac-Toe.

```
module twoinrow
Title 'Find Two Xs and an empty cell in a row, column, or diagonal'
TWOINROW device 'P22V10';

" Inputs and Outputs
X11, X12, X13, X21, X22, X23, X31, X32, X33 pin 1..9;
Y11, Y12, Y13, Y21, Y22, Y23, Y31, Y32, Y33 pin 10,11,13..15,20..23;
MOVE3..MOVE0                                pin 16..19 istype 'com';

" MOVE output encodings
MOVE   = [MOVE3..MOVE0];
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];
```

Table 6-13 (continued)

```
" Find moves in rows.  Rxy ==> a move exists in cell xy
R11 = X12 & X13 & !X11 & !Y11;
R12 = X11 & X13 & !X12 & !Y12;
R13 = X11 & X12 & !X13 & !Y13;
R21 = X22 & X23 & !X21 & !Y21;
R22 = X21 & X23 & !X22 & !Y22;
R23 = X21 & X22 & !X23 & !Y23;
R31 = X32 & X33 & !X31 & !Y31;
R32 = X31 & X33 & !X32 & !Y32;
R33 = X31 & X32 & !X33 & !Y33;

" Find moves in columns.  Cxy ==> a move exists in cell xy
C11 = X21 & X31 & !X11 & !Y11;
C12 = X22 & X32 & !X12 & !Y12;
C13 = X23 & X33 & !X13 & !Y13;
C21 = X11 & X31 & !X21 & !Y21;
C22 = X12 & X32 & !X22 & !Y22;
C23 = X13 & X33 & !X23 & !Y23;
C31 = X11 & X21 & !X31 & !Y31;
C32 = X12 & X22 & !X32 & !Y32;
C33 = X13 & X23 & !X33 & !Y33;

" Find moves in diagonals.  Dxy or Exy ==> a move exists in cell xy
D11 = X22 & X33 & !X11 & !Y11;
D22 = X11 & X33 & !X22 & !Y22;
D33 = X11 & X22 & !X33 & !Y33;
E13 = X22 & X31 & !X13 & !Y13;
E22 = X13 & X31 & !X22 & !Y22;
E31 = X13 & X22 & !X31 & !Y31;

" Combine moves for each cell.  Gxy ==> a move exists in cell xy
G11 = R11 # C11 # D11;
G12 = R12 # C12;
G13 = R13 # C13 # E13;
G21 = R21 # C21;
G22 = R22 # C22 # D22 # E22;
G23 = R23 # C23;
G31 = R31 # C31 # E31;
G32 = R32 # C32;
G33 = R33 # C33 # D33;

equations

WHEN G22 THEN MOVE= MOVE22;
ELSE WHEN G11 THEN MOVE = MOVE11;
ELSE WHEN G13 THEN MOVE = MOVE13;
ELSE WHEN G31 THEN MOVE = MOVE31;
ELSE WHEN G33 THEN MOVE = MOVE33;
ELSE WHEN G12 THEN MOVE = MOVE12;
ELSE WHEN G21 THEN MOVE = MOVE21;
ELSE WHEN G23 THEN MOVE = MOVE23;
ELSE WHEN G32 THEN MOVE = MOVE32;
ELSE MOVE = NONE;

end twoinrow
```

all possible row, column, and diagonal moves. It combines all of the moves for a cell `i,j` in an expression for `Gij`, and finally the `equations` section uses a `WHEN` statement to select a move.

Note that a nested `WHEN` statement must be used rather than nine parallel `WHEN` statements or assignments, because we can only select one move even if multiple moves are available. Also note that `G22`, the center cell, is checked first, followed by the corners. This was done hoping that we could minimize the number of terms by putting the most common moves early in the nested `WHEN`. Alas, the design still requires a ton of product terms, as shown in Table 6-14.

By the way, we still haven't explained why we chose the output coding that we did (as defined by `MOVE11`, `MOVE22`, etc. in the program). It's pretty clear that changing the encoding is never going to save us enough product terms to fit the design into a 22V10. But there's still method to this madness, as we'll now show.

Clearly we'll have to split TWOINROW into two or more pieces. As in any design problem, several different strategies are possible. The first strategy I tried was to use two different PLDs, one to find moves in all the rows and one of the diagonals, and the other to work on all the columns and the remaining diagonal. That helped, but not nearly enough to fit each half into a 22V10.

With the second strategy, I tried slicing the problem a different way. The first PLD finds all the moves in cells 11, 12, 13, 21, and 22, and the second PLD finds all the moves in the remaining cells. That worked! The first PLD, named TWOINHAF, is obtained from Table 6-13 simply by commenting out the four lines of the `WHEN` statement for the moves to cells 23, 31, 32, and 33.

We could obtain the second PLD from TWOINROW in a similar way, but let's wait a minute. In the manufacture of real digital systems, it is always desirable to minimize the number of distinct parts that are used; this saves on inventory costs and complexity. With programmable parts, it is desirable to minimize the number of distinct programs that are used. Even though the physical parts are identical, a different set of test vectors must be devised at some cost for each different program. Also, it's possible that the product will be successful enough for us to save money by converting the PLDs into hard-coded devices, a different one for each program, again encouraging us to minimize programs.

Table 6-14 Product-term usage for the TWOINROW PLD.

```
 P-Terms   Fan-in  Fan-out  Type  Name
---------  ------  -------  ----  -----------
  61/142     18        1    Pin   MOVE3
 107/129     18        1    Pin   MOVE2
  77/88      17        1    Pin   MOVE1
 133/87      18        1    Pin   MOVE0
=========
 378/446         Best P-Term Total: 332
                        Total Pins: 22
               Average P-Term/Output: 83
```

The Tic-Tac-Toe game is the same game even if we rotate the grid 90° or 180°. Thus, the TWOINHAF PLD can find moves for cells 33, 32, 31, 23, and 22 if we rotate the grid 180°. Because of the way we defined the grid state, with a separate pair of inputs for each cell, we can "rotate the grid" simply by shuffling the input pairs appropriately. That is, we swap 33↔11, 32↔12, 31↔13, and 23↔21.

Of course, once we rearrange inputs, TWOINHAF will still produce output move codes corresponding to cells in the top half of the grid. To keep things straight, we should transform these into codes for the proper cells in the bottom half of the grid. We would like this transformation to take a minimum of logic. This is where our choice of output code comes in. If you look carefully at the MOVE coding defined at the beginning of Table 6-13, you'll see that the code for a given position in the 180° rotated grid is obtained by complementing and reversing the order of the code bits for the same position in the unrotated grid. In other words, the code transformation can be done with four inverters and a rearrangement of wires. This can be done "for free" in the PLD that looks at the TWOINHAF outputs.

You probably never thought that Tic-Tac-Toe could be so tricky. Well, we're halfway there. Figure 6-13 shows the partitioning of the design as we'll now continue it. Each TWOINROW PLD from our original partition is replaced

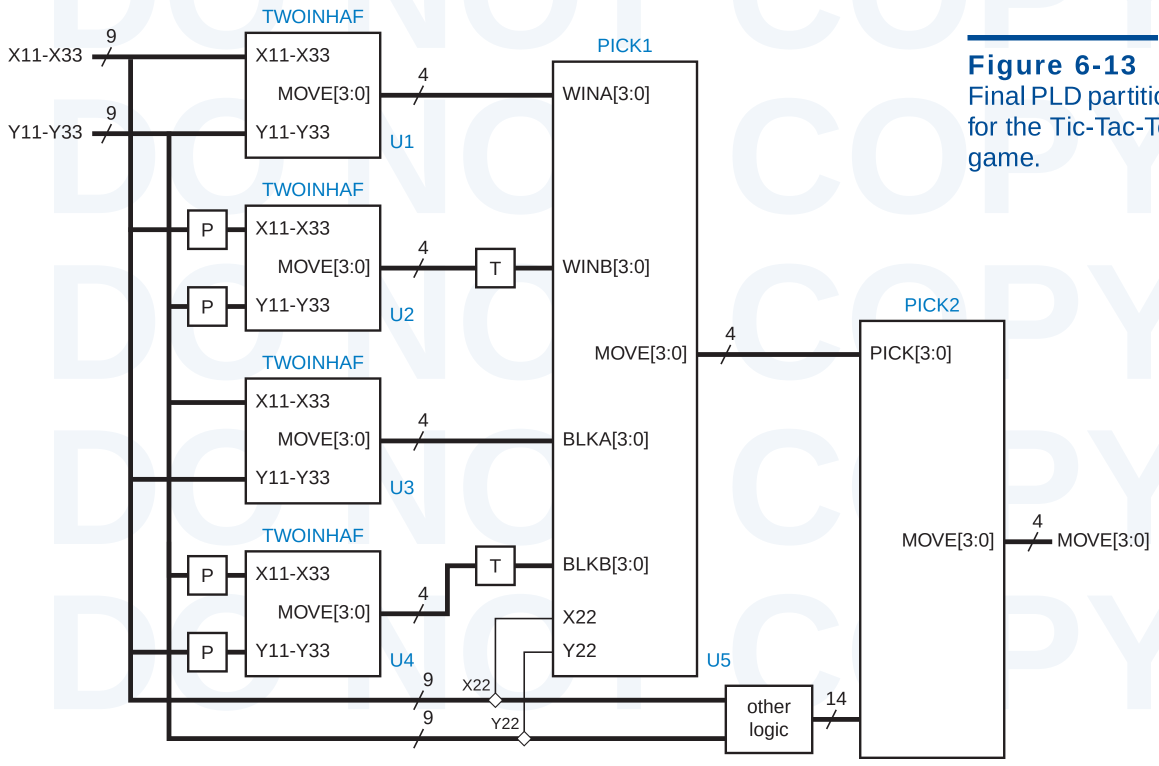


Figure 6-13 Final PLD partitioning for the Tic-Tac-Toe game.

by a pair of TWOINHALF PLDs. The bottom PLD of each pair is preceded by a box labeled “P”, which permutes the inputs to rotate the grid 180° as discussed previously. Likewise, it is followed by a box labeled “T”, which compensates for the rotation by transforming the output code; this box will actually be absorbed into the PLD that follows it, PICK1.

The function of PICK1 is pretty straightforward. As shown in Table 6-15, it is simply picks a winning move or a blocking move if one is available. Since there are two extra input pins available on the 22V10, we use them to input the state of the center cell. In this way, we can perform the first part of step 3 of the “human” algorithm on page 488, to pick the center cell if no winning or blocking move is available. The PICK1 PLD uses at most 9 product terms per output.

Table 6-15 ABEL program to pick one move based on four inputs.

```
module pick1
Title 'Pick One Move from Four Possible'
PICK1 device 'P22V10';

" Inputs from TWOINHAF PLDs
WINA3..WINA0      pin 1..4;       "Winning moves in cells 11,12,13,21,22
WINB3..WINB0      pin 5..8;       "Winning moves in cells 11,12,13,21,22 of rotated grid
BLKA3..BLKA0      pin 9..11, 13;  "Blocking moves in cells 11,12,13,21,22
BLKB3..BLKB0      pin 14..16, 21; "Blocking moves in cells 11,12,13,21,22 of rotated grid
" Inputs from grid
X22, Y22          pin 22..23;     "Center cell; pick if no other moves
" Move outputs to PICK2 PLD
MOVE3..MOVE0      pin 17..20 istype 'com';

" Sets
WINA = [WINA3..WINA0];  WINB = [WINB3..WINB0];
BLKA = [BLKA3..BLKA0];  BLKB = [BLKB3..BLKB0];
MOVE = [MOVE3..MOVE0];

" Non-rotated move input and output encoding
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];

equations

WHEN WINA != NONE THEN MOVE = WINA;
ELSE WHEN WINB != NONE THEN MOVE = ![WINB0..WINB3];  " Map rotated coding
ELSE WHEN BLKA != NONE THEN MOVE = BLKA;
ELSE WHEN BLKB != NONE THEN MOVE = ![BLKB0..BLKB3];  " Map rotated coding
ELSE WHEN !X22 & !Y22 THEN MOVE = MOVE22;            " Pick center cell if it’s empty
ELSE MOVE = NONE;

end pick1
```

The final part of the design in Figure 6-13 is the PICK2 PLD. This PLD must provide most of the “experience” in step 3 of the human algorithm if PICK1 does not find a move.

We have a little problem with PICK2 in that a 22V10 does not have enough pins to accommodate the 4-bit input from PICK1, its own 4-bit output, and all 18 bits of grid state; it has only 22 I/O pins. Actually, we don’t need to connect X22 and Y22, since they were already examined in PICK1, but that still leaves us two pins short. So, the purpose of the “other logic” block in Figure 6-13 is to encode

Table 6-16 ABEL program to pick one move using “experience.”

```
module pick2
Title 'Pick a move using experience'
PICK2 device 'P22V10';

" Inputs from PICK1 PLD
PICK3..PICK0                              pin 1..4;  " Move, if any, from PICK1 PLD
" Inputs from Tic-Tac-Toe grid corners
X11, Y11, X13, Y13, X31, Y31, X33, Y33   pin 5..11, 13;
" Combined inputs from external NOR gates; 1 ==> corresponding cell is empty
E12, E21, E23, E32                        pin 14..15, 22..23;
" Move output
MOVE3..MOVE0                              pin 17..20 istype 'com';

PICK = [PICK3..PICK0];  " Set definition
" Non-rotated move input and output encoding
MOVE = [MOVE3..MOVE0];
MOVE11 = [1,0,0,0]; MOVE12 = [0,1,0,0]; MOVE13 = [0,0,1,0];
MOVE21 = [0,0,0,1]; MOVE22 = [1,1,0,0]; MOVE23 = [0,1,1,1];
MOVE31 = [1,0,1,1]; MOVE32 = [1,1,0,1]; MOVE33 = [1,1,1,0];
NONE   = [0,0,0,0];

" Intermediate equations for empty corner cells
E11 = !X11 & !Y11;  E13 = !X13 & !Y13;  E31 = !X31 & !Y31;  E33 = !X33 & !Y33;

equations

"Simplest approach -- pick corner if available, else side
WHEN PICK != NONE THEN MOVE = PICK;
ELSE WHEN E11 THEN MOVE = MOVE11;
ELSE WHEN E13 THEN MOVE = MOVE13;
ELSE WHEN E31 THEN MOVE = MOVE31;
ELSE WHEN E33 THEN MOVE = MOVE33;
ELSE WHEN E12 THEN MOVE = MOVE12;
ELSE WHEN E21 THEN MOVE = MOVE21;
ELSE WHEN E23 THEN MOVE = MOVE23;
ELSE WHEN E32 THEN MOVE = MOVE32;
ELSE MOVE = NONE;

end pick2
```

some of the information to save two pins. The method that we'll use here is to combine the signals for the middle edge cells 12, 21, 23, and 32 to produce four signals E12, E21, E23, and E32 that are asserted if and only if the corresponding cells are empty. This can be done with four 2-input NOR gates, and actually leaves two spare inputs or outputs on the 22V10.

Assuming the four NOR gates as "other logic," Table 6-16 on the preceding page gives a program for the PICK2 PLD. When it must pick a move, this program uses the simplest heuristic possible—it picks a corner cell if one is empty, else it picks a middle edge cell. This program could use some improvement, because it will sometimes lose (see Exercise 6.8). Luckily, the equations resulting from Table 6-16 require only 8 to 10 terms per output, so it's possible to put in more intelligence (see Exercises 6.9 and 6.10).

6.3 Design Examples Using VHDL

6.3.1 Barrel Shifter

On page 464, we defined a barrel shifter as a combinational logic circuit with *n* data inputs, *n* data outputs, and a set of control inputs that specify how to shift the data between input and output. We showed in Section 6.1.1 how to build a simple barrel shifter that performs only left circular shifts using MSI building blocks. Later, in Section 6.2.1, we showed how to define a more capable barrel shifter using ABEL, but we also pointed out that PLDs are normally unsuitable for realizing barrel shifters. In this subsection, we'll show how VHDL can be used to describe both the behavior and structure of barrel shifters for FPGA or ASIC realization.

Table 6-17 is a behavioral VHDL program for a 16-bit barrel shifter that performs any of six different combinations of shift type and direction. The shift types are circular, logical, and arithmetic, as defined previously in Table 6-3, and the directions are of course left and right. As shown in the entity declaration, a 4-bit control input `S` gives the shift amount, and a 3-bit control input `C` gives the shift mode (type and direction). We used the IEEE `std_logic_arith` package and defined the shift amount `S` to be type `UNSIGNED` so we could later use the `CONV_INTEGER` function in that package.

Notice that the entity declaration includes six constant definitions that establish the correspondence between shift modes and the value of `C`. Although we didn't discuss it in Section 4.7, VHDL allows you to put constant, type, signal, and other declarations within an entity declaration. It makes sense to define such items within the entity declaration only if they must be the same in any architecture. In this case, we are pinning down the shift-mode encodings, so they should go here. Other items should go in the architecture definition.

In the architecture part of the program, we define six functions, one for each kind of shift on a 16-bit `STD_LOGIC_VECTOR`. We defined the subtype `DATAWORD` to save typing in the function definitions.

Table 6-17 VHDL behavioral description of a 6-function barrel shifter.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity barrel16 is
    port (
        DIN: in STD_LOGIC_VECTOR (15 downto 0);  -- Data inputs
        S: in UNSIGNED (3 downto 0);             -- Shift amount, 0-15
        C: in STD_LOGIC_VECTOR (2 downto 0);     -- Mode control
        DOUT: out STD_LOGIC_VECTOR (15 downto 0) -- Data bus output
    );
    constant Lrotate:  STD_LOGIC_VECTOR := "000";   -- Define the coding of
    constant Rrotate:  STD_LOGIC_VECTOR := "001";   -- the different shift modes
    constant Llogical: STD_LOGIC_VECTOR := "010";
    constant Rlogical: STD_LOGIC_VECTOR := "011";
    constant Larith:   STD_LOGIC_VECTOR := "100";
    constant Rarith:   STD_LOGIC_VECTOR := "101";
end barrel16;

architecture barrel16_behavioral of barrel16 is
subtype DATAWORD is STD_LOGIC_VECTOR(15 downto 0);

function Vrol (D: DATAWORD; S: UNSIGNED)
     return DATAWORD is
  variable N: INTEGER;
  variable TMPD: DATAWORD;
  begin
    N := CONV_INTEGER(S); TMPD := D;
    for i in 1 to N loop
      TMPD := TMPD(14 downto 0) & TMPD(15);
    end loop;
    return TMPD;
  end Vrol;
...

begin
process(DIN, S, C)
  begin
    case C is
      when Lrotate  => DOUT <= Vrol(DIN,S);
      when Rrotate  => DOUT <= Vror(DIN,S);
      when Llogical => DOUT <= Vsll(DIN,S);
      when Rlogical => DOUT <= Vsrl(DIN,S);
      when Larith   => DOUT <= Vsla(DIN,S);
      when Rarith   => DOUT <= Vsra(DIN,S);
      when others   => null;
    end case;
  end process;
end barrel16_behavioral;
```

ROLLING YOUR OWN

VHDL-93 actually has built-in array operators, `rol`, `ror`, `sll`, `srl`, `sla`, and `sra`, corresponding to the shift operations that we defined in Table 6-3. Since these operations are not provided in VHDL-87, we've defined our own functions in Table 6-17. Well, actually we've only defined one of them (`Vrol`); the rest are left as an exercise for the reader (Exercise 6.11).

Table 6-17 shows the details of only the first function (`Vrol`); the rest are similar with only a one-line change. We define a variable `N` for converting the shift amount `S` into an integer for the `for` loop. We also assign the input vector `D` to a local variable `TMPD` which is shifted `N` times in the `for` loop. In the body of the `for` loop, a single assignment statement takes a 15-bit slice of the data word (`TMPD 14 downto 0)`) and uses concatenation (`&`) to put it back together with the bit that "falls off" the left end (`TMPD(15)`). Other shift types can be described with similar operations. Note that the shift functions might not be defined in other, nonbehavioral descriptions of the `barrel16` entity, for example in structural architectures.

The "concurrent statements" part of the architecture is a single process that has all of the entity's inputs in its sensitivity list. Within this process, a `case` statement assigns a result to `DOUT` by calling the appropriate function based on the value of the mode-control input `C`.

The process in Table 6-17 is a nice behavioral description of the barrel shifter, but most synthesis tools cannot synthesize a circuit from it. The problem is that most tools require the range of a `for` loop to be static at the time it is analyzed. The range of the `for` loop in the `Vrol` function is dynamic; it depends on the value of input signal `S` when the circuit is operating.

Well, that's OK, it's hard to predict what kind of circuit the synthesis tool would come up with even if it could handle a dynamic `for` range. This is an example where as designers we should take little more control over the circuit structure to obtain a reasonably fast, efficient synthesis result.

In Figure 6-2 on page 466, we showed how to design a 16-bit barrel shifter for left circular shifts using MSI building blocks. We used a cascade of four 16-bit, 2-input multiplexers to shift their inputs by 0 or 1, 2, 4, or 8 positions depending of the values of S0 through S3, respectively. We can express the same kind behavior and structure using the VHDL program shown in Table 6-18. Even though the program uses a process and is therefore "behavioral" in style, we can be pretty sure that most synthesis engines will generate a 1-input multiplexer for each "`if`" statement in the program, thereby creating a cascade similar to Figure 6-2.

Table 6-18 VHDL program for a 16-bit barrel shifter for left circular shifts only.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rol16 is
    port (
        DIN: in STD_LOGIC_VECTOR(15 downto 0);  -- Data inputs
        S: in STD_LOGIC_VECTOR (3 downto 0);    -- Shift amount, 0-15
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );
end rol16;

architecture rol16_arch of rol16 is
begin
process(DIN, S)
  variable X, Y, Z: STD_LOGIC_VECTOR(15 downto 0);
  begin
    if S(0)='1' then X := DIN(14 downto 0) & DIN(15); else X := DIN; end if;
    if S(1)='1' then Y := X(13 downto 0) & X(15 downto 14); else Y := X; end if;
    if S(2)='1' then Z := Y(11 downto 0) & Y(15 downto 12); else Z := Y; end if;
    if S(3)='1' then DOUT <= Z(7 downto 0) & Z(15 downto 8); else DOUT <= Z; end if;
  end process;
end rol16_arch;
```

Of course, our problem statement requires a barrel shifter that can shift both left and right. Table 6-19 revises the previous program to do circular shifts in either direction. An additional input, `DIR`, specifies the shift direction, 0 for left, 1 for right. Each rank of shifting is specified by a `case` statement that picks one of four possibilities based on the values of `DIR` and the bit of `S` that controls that rank. Notice that we created local 2-bit variables `CTRLi` to hold the pair of values `DIR` and `S(i)`; each `case` statement is controlled by one of these variables. You might like to eliminate these variables and simply control each case statement with a concatenation "`DIR & S(i)`", but VHDL syntax doesn't allow that because the type of this concatenation would be unknown.

A typical VHDL synthesis tool will generate a 3- or 4-input multiplexer for each of the `case` statements in Table 6-19. A good synthesis tool will generate only a 2-input multiplexer for the last `case` statement.

So, now we have a barrel shifter that will do left or right circular shifts, but we're not done yet—we need to take care of the logical and arithmetic shifts in both directions. Figure 6-14 shows our strategy for completing the design. We start out with the `ROLR16` component that we just completed, and use other logic to control the shift direction as a function of `C`.

Table 6-19 VHDL program for a 16-bit barrel shifter for left and right circular shifts.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rolr16 is
    port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0);   -- Data inputs
        S:    in STD_LOGIC_VECTOR (3 downto 0);   -- Shift amount, 0-15
        DIR:  in STD_LOGIC;                       -- Shift direction, 0=>L, 1=>R
        DOUT: out STD_LOGIC_VECTOR(15 downto 0)   -- Data bus output
    );
end rolr16;

architecture rol16r_arch of rolr16 is
begin
process(DIN, S, DIR)
  variable X, Y, Z: STD_LOGIC_VECTOR(15 downto 0);
  variable CTRL0, CTRL1, CTRL2, CTRL3: STD_LOGIC_VECTOR(1 downto 0);
  begin
    CTRL0 := S(0) & DIR; CTRL1 := S(1) & DIR; CTRL2 := S(2) & DIR; CTRL3 := S(3) & DIR;
    case CTRL0 is
      when "00" | "01" => X := DIN;
      when "10" => X := DIN(14 downto 0) & DIN(15);
      when "11" => X := DIN(0) & DIN(15 downto 1);
      when others => null;  end case;
    case CTRL1 is
      when "00" | "01" => Y := X;
      when "10" => Y := X(13 downto 0) & X(15 downto 14);
      when "11" => Y := X(1 downto 0) & X(15 downto 2);
      when others => null;  end case;
    case CTRL2 is
      when "00" | "01" => Z := Y;
      when "10" => Z := Y(11 downto 0) & Y(15 downto 12);
      when "11" => Z := Y(3 downto 0) & Y(15 downto 4);
      when others => null;  end case;
    case CTRL3 is
      when "00" | "01" => DOUT <= Z;
      when "10" | "11" => DOUT <= Z(7 downto 0) & Z(15 downto 8);
      when others => null;  end case;
  end process;
end rol16r_arch;
```

Next we must "fix up" some of the result bits if we are doing a logical or arithmetic shift. For a left logical or arithmetic *n*-bit shift, we must set the rightmost *n*–1 bits to 0 or the original rightmost bit value, respectively. For a right logical or arithmetic *n*-bit shift, we must set the leftmost *n*–1 bits to 0 or the original leftmost bit value, respectively.

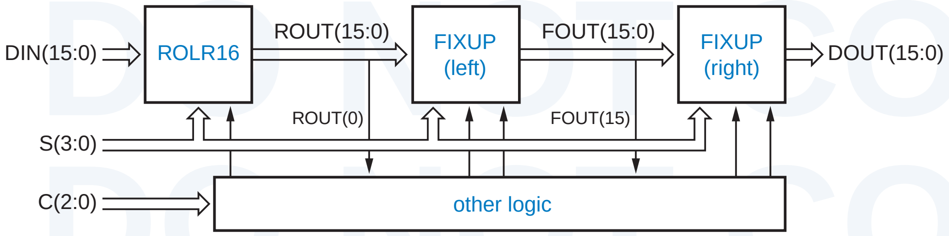


Figure 6-14 Barrel-shifter components.

As shown in Figure 6-14, our strategy is to follow the circular shifter (ROLR16) with a fix-up circuit (FIXUP) that plugs in appropriate low-order bits for a left logical or arithmetic shift, and follow that with another fix-up circuit that plugs in high-order bits for a right logical or arithmetic shift.

Table 6-20 is a behavioral VHDL program for the left-shift fix-up circuit. The circuit has 16 bits of data input and output, `DIN` and `DOUT`. Its control inputs are the shift amount `S`, an enable input `FEN`, and the new value `FDAT` to be plugged into the fixed-up data bits. For each output bit `DOUT(i)`, the circuit puts out the fixed-up bit value if `i` is less than `S` and the circuit is enabled; else it puts out the unmodified data input `DIN(i)`.

The `for` loop in Table 6-20 is readily synthesizable, but you can’t really be sure what kind of logic the synthesis tool will generate. In particular, the “>”

Table 6-20 Behavioral VHDL program for left-shift fix-ups.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity fixup is
    port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0);          -- Shift amount, 0-15
        FEN:  in STD_LOGIC;                     -- Fixup enable
        FDAT: in STD_LOGIC;                     -- Fixup data
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );
end fixup;

architecture fixup_arch of fixup is
begin
process(DIN, S, FEN, FDAT)
  begin
    for i in 0 to 15 loop
      if (i < CONV_INTEGER(S)) and (FEN = '1') then DOUT(i) <= FDAT;
      else DOUT(i) <= DIN(i);  end if;
    end loop;
  end process;
end fixup_arch;
```

A SERIAL FIX-UP STRUCTURE

A structural architecture for the fix-up logic is shown in Table 6-21. Here, we have defined what is in effect an iterative circuit to create a 16-bit vector `FSEL`, where `FSEL(i)` is 1 if bit `i` needs fixing up. We start by setting `FSEL(15)` to 0, since that bit never needs fixing up. Then we note that for the remaining values of `i`, `FSEL(i)` should be 1 if `S` equals `i+1` or if `FSEL(i+1)` is already asserted. Thus, the `FSEL` assignment within the generate statement creates a serial chain of 2-input OR gates, where one input is asserted if `S=i` (decoded with a 4-input AND gate), and the other input is connected to the previous OR gate's output. The `DOUT(i)` assignment statement creates 16 2-input multiplexers that select either `DIN(i)` or the fix-up data (`FDAT`) depending of the value of `FSEL(i)`.

Although the serial realization is compact, it is very slow compared to a one that realizes each `FSEL` output as a 2-level sum-of-products circuit. However, the long delay may not matter because the fix-up circuit appears near the end of the data path. If speed is still a problem, there is a zero-cost trick that cuts the delay in half (see Exercise 6.12).

operation in each step of the loop may cause the synthesis of a general-purpose magnitude comparator, even though one of the operands is a constant and each output could therefore be generated with no more than a handful of gates. (In fact, the logic for "7 < `CONV_INTEGER(S)`" is just a wire, `S(3)`!) For a structural version of this function, see the box on this page.

For right shifts, fix-ups start from the opposite end of the data word, so it would appear that we need a second version of the fix-up circuit. However, we can use the original version if we just reverse the order of its input and output bits, as we'll soon see.

Table 6-22 puts together a structural architecture for the complete, 16-bit, 6-function barrel shifter using the design approach of Figure 6-14 on page 501. The entity declaration for barrel16 is unchanged from the original in Table 6-17 on page 497. The architecture declares two components, `rolr16` and `fixup`; these use our previous entity definitions. The statement part of the architecture

Table 6-21 Structural VHDL architecture for left-shift fix-ups.

```
architecture fixup_struc of fixup is
signal FSEL: STD_LOGIC_VECTOR(15 downto 0);      -- Fixup select
begin
  FSEL(15) <= '0'; DOUT(15) <= DIN(15);
  U1: for i in 14 downto 0 generate
    FSEL(i) <= '1' when CONV_INTEGER(S) = i+1 else FSEL(i+1);
    DOUT(i) <= FDAT when (FSEL(i) = '1' and FEN = '1') else DIN(i);
  end generate;
end fixup_struc;
```

instantiates rolr16 and fixup and has several assignment statements that create needed control signals (the “other logic” in Figure 6-14).

For example, the first assignment statement asserts DIR_RIGHT if C specifies one of the right shifts. The enable inputs for the left and right fix-up circuits are FIX_LEFT and FIX_RIGHT, asserted for left and right logical and arithmetic shifts. The fix-up data values are FIX_LEFT_DAT and FIX_RIGHT_DAT.

While all the statements in the architecture execute concurrently, they are listed in Table 6-22 in the order of the actual dataflow to improve readability. First, rolr16 is instantiated to perform the basic left or right circular shift as specified. Its outputs are hooked up to the inputs of the first fixup component (U2) to handle fix-ups for left logical and arithmetic shifts. Next comes U3, a generate statement that reverses the order of the data inputs for the next fixup

Table 6-22 VHDL structural architecture for the 6-function barrel shifter.

```
architecture barrel16_struc of barrel16 is

component rolr16 port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0);          -- Shift amount, 0-15
        DIR:  in STD_LOGIC;                     -- Shift direction, 0=>L, 1=>R
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );  end component;

component fixup port (
        DIN:  in STD_LOGIC_VECTOR(15 downto 0); -- Data inputs
        S:    in UNSIGNED(3 downto 0);          -- Shift amount, 0-15
        FEN:  in STD_LOGIC;                     -- Fixup enable
        FDAT: in STD_LOGIC;                     -- Fixup data
        DOUT: out STD_LOGIC_VECTOR(15 downto 0) -- Data bus output
    );  end component;

signal DIR_RIGHT, FIX_RIGHT, FIX_RIGHT_DAT, FIX_LEFT, FIX_LEFT_DAT: STD_LOGIC;
signal ROUT, FOUT, RFIXIN, RFIXOUT: STD_LOGIC_VECTOR(15 downto 0);

begin
  DIR_RIGHT <= '1' when C = Rrotate or C = Rlogical or C = Rarith else '0';
  FIX_LEFT <= '1' when DIR_RIGHT='0' and (C = Llogical or C = Larith) else '0';
  FIX_RIGHT <= '1' when DIR_RIGHT='1' and (C = Rlogical or C = Rarith) else '0';
  FIX_LEFT_DAT <= DIN(0) when C = Larith else '0';
  FIX_RIGHT_DAT <= DIN(15) when C = Rarith else '0';
  U1: rolr16 port map (DIN, S, DIR_RIGHT, ROUT);
  U2: fixup port map (ROUT, S, FIX_LEFT, FIX_LEFT_DAT, FOUT);
  U3: for i in 0 to 15 generate RFIXIN(i) <= FOUT(15-i); end generate;
  U4: fixup port map (RFIXIN, S, FIX_RIGHT, FIX_RIGHT_DAT, RFIXOUT);
  U5: for i in 0 to 15 generate DOUT(i) <= RFIXOUT(15-i); end generate;
end barrel16_struc;
```

INFORMATION-HIDING STYLE

Based on the encoding of C, you might like to replace the first assignment statement in Table 6-21 with "`DIR_RIGHT <= C(0)`", which would be guaranteed to lead to a more efficient realization for that control bit—just a wire! However, this would violate a programming principle of information hiding and lead to possible bugs.

We explicitly wrote the shift encodings using `constant` definitions in the `barrel16` entity declaration. The architecture does not need to be aware of the encoding details. Suppose that we nevertheless made the architecture change suggested above. If somebody else (or we!) came along later and changed the `constant` definitions in the `barrel16` entity to make a different encoding, the architecture would not use the new encodings! Exercise 6.13 asks you to change the definitions so that the cost savings of our suggested change are enabled by the entity definition.

component (`U4`), which handles fix-ups for right logical and arithmetic shifts. Finally `U5`, another `generate` statement, undoes the bit reversing of `U3`. Note that in synthesis, `U3` and `U5` are merely permutations of wires.

Many other architectures are possible for the original `barrel16` entity. In Exercise 6.14, we suggest an architecture that enables the circular shifting to be done by the `rol16` entity, which uses only 2-input multiplexers, rather than the more expensive `rolr16`.

6.3.2 Simple Floating-Point Encoder

We defined a simple floating-point number format on page 467, and posed the design problem of converting a number from fixed-point to this floating point format. The problem of determining the exponent of the floating-point number mapped nicely into an MSI priority encoder. In an HDL, the same problem maps into nested "`if`" statements.

Table 6-23 is a behavioral VHDL program for the floating-point encoder. Within the `fpenc_arch` architecture, a nested "`if`" statement checks the range of the input `B` and sets `M` and `E` appropriately. Notice that the program uses the IEEE `std_logic_arith` package; this is done to get the `UNSIGNED` type and the comparison operations that go along with it, as we described in Section 5.9.6. Just to save typing, a variable `BU` is defined to hold the value of B as converted to the `UNSIGNED` type; alternatively, we could have written "`UNSIGNED(B)`" in each nested "`if`" clause.

Although the code in Table 6-23 is fully synthesizable, some synthesis tools may not be smart enough to recognize that the nested comparisons require just one bit to be checked at each level, and might instead generate a full 11-bit comparator at each level. Such logic would be a lot bigger and slower than what would be otherwise possible. If faced with this problem, we can always write the architecture a little differently and more explicitly to help out the tool, as shown in Table 6-24.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity fpenc is
  port (
    B: in STD_LOGIC_VECTOR(10 downto 0); -- fixed-point number
    M: out STD_LOGIC_VECTOR(3 downto 0); -- floating-point mantissa
    E: out STD_LOGIC_VECTOR(2 downto 0)  -- floating-point exponent
  );
end fpenc;

architecture fpenc_arch of fpenc is
begin
  process(B)
  variable BU: UNSIGNED(10 downto 0);
  begin
    BU := UNSIGNED(B);
    if    BU < 16   then M <= B( 3 downto 0); E <= "000";
    elsif BU < 32   then M <= B( 4 downto 1); E <= "001";
    elsif BU < 64   then M <= B( 5 downto 2); E <= "010";
    elsif BU < 128  then M <= B( 6 downto 3); E <= "011";
    elsif BU < 256  then M <= B( 7 downto 4); E <= "100";
    elsif BU < 512  then M <= B( 8 downto 5); E <= "101";
    elsif BU < 1024 then M <= B( 9 downto 6); E <= "110";
    else                 M <= B(10 downto 7); E <= "111";
    end if;
  end process;
end fpenc_arch;
```

Table 6-23 Behavioral VHDL program for fixed-point to floating-point conversion.

```
architecture fpence_arch of fpenc is
begin
  process(B)
  begin
    if   B(10) = '1' then M <= B(10 downto 7); E <= "111";
    elsif B(9) = '1' then M <= B( 9 downto 6); E <= "110";
    elsif B(8) = '1' then M <= B( 8 downto 5); E <= "101";
    elsif B(7) = '1' then M <= B( 7 downto 4); E <= "100";
    elsif B(6) = '1' then M <= B( 6 downto 3); E <= "011";
    elsif B(5) = '1' then M <= B( 5 downto 2); E <= "010";
    elsif B(4) = '1' then M <= B( 4 downto 1); E <= "001";
    else                  M <= B( 3 downto 0); E <= "000";
    end if;
  end process;
end fpence_arch;
```

Table 6-24 Alternative VHDL architecture for fixed-point to floating-point conversion.

B'S NOT MY TYPE

In Table 6-23, we used the expression `UNSIGNED(B)` to convert `B`, an array of type `STD_LOGIC_VECTOR`, into an array of type `UNSIGNED`. This is called an *explicit type conversion*. VHDL lets you convert between related closely related types by writing the desired type followed by the value to be converted in parentheses. Two array types are "closely related" if they have the same element type, the same number of dimensions, and the same index types (typically `INTEGER`) or ones that can be type converted. The values in the old array are placed in corresponding positions, left to right, in the new array.

On the other hand, we might like to use the real comparators and spend even more gates to improve the functionality of our design. In particular, the present design performs truncation rather than rounding when generating the mantissa bits. A more accurate result is achieved with rounding, but this is a much more complicated design. First, we will need an adder to add 1 to the selected mantissa bits when we round up. However, adding 1 when the mantissa is already 1111 will bump us into the next exponent range, so we need to watch out for this case. Finally, we can never round up if the unrounded mantissa and exponent are 1111 and 111, because there's no higher value in our floating-point representation to round to.

The program in Table 6-25 performs rounding as desired. The function `round` takes a selected 5-bit slice from the fixed-point number and returns the four high-order bits, adding 1 if the LSB is 1. Thus, if we think of the binary point as being just to the left of the LSB, rounding occurs if the truncated part of the mantissa is 1/2 or more. In each clause in the nested "`if`" statement in the process, the comparison value is selected so that rounding up will occur only if it does not "overflow," pushing the result into the next exponent range. Otherwise, conversion and rounding occurs in the next clause. In the last clause, we ensure that we do not round up when we're at the end of the floating-point range.

GOBBLE, GOBBLE

The rounding operation does not require a 4-bit adder, only an "incrementer," since one of the addends is always 1. Some VHDL tools may synthesize the complete adder, while others may be smart enough to use an incrementer with far fewer gates.

In some cases, it may not matter. The most sophisticated tools for FPGA and ASIC design include *gate gobblers*. These programs look for gates with constant inputs and eliminate gates or gate inputs as a result. For example, an AND-gate input with a constant 1 applied to it can be eliminated, and an AND gate with a constant-0 input can be replaced with a constant-0 signal.

A gate-gobbler program propagates the effects of constant inputs as far as possible in a circuit. Thus, it can transform a 4-bit adder with a constant-1 input into a more economical 4-bit incrementer.

```
architecture fpencr_arch of fpenc is
function round (BSLICE: STD_LOGIC_VECTOR(4 downto 0))
    return STD_LOGIC_VECTOR is
  variable BSU: UNSIGNED(3 downto 0);
  begin
    if BSLICE(0) = '0' then return BSLICE(4 downto 1);
    else null;
      BSU := UNSIGNED(BSLICE(4 downto 1)) + 1;
      return STD_LOGIC_VECTOR(BSU);
    end if;
  end;
begin
  process(B)
  variable BU: UNSIGNED(10 downto 0);
  begin
    BU := UNSIGNED(B);
    if    BU < 16      then M <= B( 3 downto 0); E <= "000";
    elsif BU < 32-1    then M <= round(B( 4 downto 0)); E <= "001";
    elsif BU < 64-2    then M <= round(B( 5 downto 1)); E <= "010";
    elsif BU < 128-4   then M <= round(B( 6 downto 2)); E <= "011";
    elsif BU < 256-8   then M <= round(B( 7 downto 3)); E <= "100";
    elsif BU < 512-16  then M <= round(B( 8 downto 4)); E <= "101";
    elsif BU < 1024-32 then M <= round(B( 9 downto 5)); E <= "110";
    elsif BU < 2048-64 then M <= round(B(10 downto 6)); E <= "111";
    else                    M <= "1111";                E <= "111";
    end if;
  end process;
end fpencr_arch;
```

Table 6-25 Behavioral VHDL architecture for fixed-point to floating-point conversion with rounding.

Once again, synthesis results for this behavioral program may or may not be efficient. Besides the multiple comparison statements, we now must worry about the multiple 4-bit adders that might be synthesized as a result of the multiple calls to the round `function`. Restructuring the architecture so that only a single adder is synthesized is left as an exercise (6.15).

6.3.3 Dual-Priority Encoder

In this example, we'll use VHDL to create a behavioral description of a PLD priority encoder that identifies both the highest-priority and the second-highest-priority asserted signal among a set of request inputs `R(0 to 7)`, where `R(0)` has the highest priority. We'll use `A(2 downto 0)` and `AVALID` to identify the highest-priority request, asserting `AVALID` only if a highest-priority request is present. Similarly, we'll use `B(2 downto 0)` and `BVALID` to identify the second-highest-priority request.

Table 6-26 Behavioral VHDL program for a dual priority encoder.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Vprior2 is
    port (
        R: in STD_LOGIC_VECTOR (0 to 7);
        A, B: out STD_LOGIC_VECTOR (2 downto 0);
        AVALID, BVALID: buffer STD_LOGIC
    );
end Vprior2;

architecture Vprior2_arch of Vprior2 is
begin
  process(R, AVALID, BVALID)
  begin
    AVALID <= '0'; BVALID <= '0'; A <= "000"; B <= "000";
    for i in 0 to 7 loop
      if R(i) = '1' and AVALID = '0' then
        A <= CONV_STD_LOGIC_VECTOR(i,3); AVALID <= '1';
      elsif R(i) = '1' and BVALID = '0' then
        B <= CONV_STD_LOGIC_VECTOR(i,3); BVALID <= '1';
      end if;
    end loop;
  end process;
end Vprior2_arch;
```

Table 6-26 shows a behavioral VHDL program for the priority encoder. Instead of the nested “`if`” approach of the previous example, we’ve used a “`for`” loop. This approach allows us to take care of both the first and the second priorities within the same loop, working our way from highest to lowest priority. Besides `std_logic_1164`, the program uses the IEEE `std_logic_arith` package in order to get the `CONV_STD_LOGIC_VECTOR` function. We also wrote this function explicitly in Table 4-39 on page 275.

Notice in the table that ports `AVALID` and `BVALID` are declared as mode `buffer`, because they are read within the architecture. If you were stuck with an entity definition that declared `AVALID` and `BVALID` as mode `out`, you could still use the same architecture approach, but you would have to declare local variables corresponding to `AVALID` and `BVALID` within the process. Notice also that we included `AVALID` and `BVALID` in the process sensitivity list. Although this is not strictly necessary, it prevents warnings that the compiler otherwise would give about using the value of a signal that is not on the sensitivity list.

The nested “`if`” approach can also be used for the dual-priority encoder, but it yields a longer program with more accidents waiting to happen, as shown in Table 6-27. On the other hand, it may yield a better synthesis result; the only

Table 6-27 Alternative VHDL architecture for a dual priority encoder.

```
architecture Vprior2i_arch of Vprior2 is
begin
  process(R, A, AVALID, BVALID)
  begin
    if    R(0) = '1' then A <= "000"; AVALID <= '1';
    elsif R(1) = '1' then A <= "001"; AVALID <= '1';
    elsif R(2) = '1' then A <= "010"; AVALID <= '1';
    elsif R(3) = '1' then A <= "011"; AVALID <= '1';
    elsif R(4) = '1' then A <= "100"; AVALID <= '1';
    elsif R(5) = '1' then A <= "101"; AVALID <= '1';
    elsif R(6) = '1' then A <= "110"; AVALID <= '1';
    elsif R(7) = '1' then A <= "111"; AVALID <= '1';
    else                  A <= "000"; AVALID <= '0';
    end if;
    if    R(1) = '1' and A /= "001" then B <= "001"; BVALID <= '1';
    elsif R(2) = '1' and A /= "010" then B <= "010"; BVALID <= '1';
    elsif R(3) = '1' and A /= "011" then B <= "011"; BVALID <= '1';
    elsif R(4) = '1' and A /= "100" then B <= "100"; BVALID <= '1';
    elsif R(5) = '1' and A /= "101" then B <= "101"; BVALID <= '1';
    elsif R(6) = '1' and A /= "110" then B <= "110"; BVALID <= '1';
    elsif R(7) = '1' and A /= "111" then B <= "111"; BVALID <= '1';
    else                                 B <= "000"; BVALID <= '0';
    end if;
  end process;
end Vprior2i_arch;
```

way to know with a particular tool is to synthesize the circuit and analyze the results in terms of delay and cell or gate count.

Both nested "`if`" statements and "`for`" statements may lead to long delay chains in synthesis. To get guarantee that you get a faster dual-priority encoder, you must follow a structural or semi-structural design approach. For example, you can start by writing a dataflow model of a fast 8-input priority encoder using the ideas found in the 74x148 logic diagram (Figure 5-50 on page 375) or in a related ABEL program (Table 5-24 on page 378). Then you can put two of these together in a structure that "knocks out" the highest-priority input in order to find the second, as we showed in Figure 6-6 on page 471.

6.3.4 Cascading Comparators

Cascading comparators is something we typically would not do in a VHDL behavioral model, because the language and the IEEE `std_logic_arith` package let us define comparators of any desired length directly. However, we may indeed need to write structural or semi-structural VHDL programs that hook up smaller comparator components in a specific way to obtain high performance.

Table 6-28
Behavioral VHDL program for a 64-bit comparator.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity comp64 is
    port ( A, B: in STD_LOGIC_VECTOR (63 downto 0);
           EQ, GT: out STD_LOGIC );
end comp64;

architecture comp64_arch of comp64 is
begin
  EQ <= '1' when A = B else '0';
  GT <= '1' when A > B else '0';
end comp64_arch;
```

Table 6-28 is a simple behavioral model of a 64-bit comparator with equals and greater-than outputs. This program uses the IEEE `std_logic_unsigned` package, whose built-in comparison functions automatically treat all signals of type `STD_LOGIC_VECTOR` as unsigned integers. Although the program is fully synthesizable, the speed and size of the result depends on the “intelligence” of the particular tool that is used.

An alternative is to build the comparator by cascading smaller components, such as 8-bit comparators. Table 6-29 is the behavioral model of an 8-bit comparator. A particular tool may or may not synthesize a very fast comparator from this program, but it’s sure to be significantly faster than a 64-bit comparator in any case.

Next, we can write a structural program that instantiates eight of these 8-bit comparators and hooks up their individual outputs through additional logic to calculate the overall comparison result. One way to do this is shown Table 6-30. A `generate` statement creates not only the individual 8-bit comparators, but

Table 6-29
VHDL program for an 8-bit comparator.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity comp8 is
    port ( A, B: in STD_LOGIC_VECTOR (7 downto 0);
           EQ, GT: out STD_LOGIC );
end comp8;

architecture comp8_arch of comp8 is
begin
  EQ <= '1' when A = B else '0';
  GT <= '1' when A > B else '0';
end comp8_arch;
```

Table 6-30 VHDL structural architecture for a 64-bit comparator.

```
architecture comp64s_arch of comp64 is
component comp8
  port ( A, B: in STD_LOGIC_VECTOR (7 downto 0);
         EQ, GT: out STD_LOGIC);
end component;
signal EQ8, GT8: STD_LOGIC_VECTOR (7 downto 0); -- =, > for 8-bit slice
signal SEQ, SGT: STD_LOGIC_VECTOR (8 downto 0); -- serial chain of slice results
begin
  SEQ(8) <= '1'; SGT(8) <= '0';
  U1: for i in 7 downto 0 generate
        U2: comp8 port map (A(7+i*8 downto i*8), B(7+i*8 downto i*8), EQ8(i), GT8(i));
        SEQ(i) <= SEQ(i+1) and EQ8(i);
        SGT(i) <= SGT(i+1) or (SEQ(i+1) and GT8(i));
  end generate;
  EQ <= SEQ(0); GT <= SGT(0);
end comp64s_arch;
```

also cascading logic that serially builds up the overall result from most significant to least significant stage.

An unsophisticated tool could synthesize a slow iterative comparator circuit for our original 64-bit comparator architecture in Table 6-28. In this situation, the architecture in Table 6-30 yields a faster synthesized circuit because it explicitly “pulls” out the cascading information for each 8-bit slice and combines it in a faster combinational circuit (just 8 levels of AND-OR logic, not 64). A more sophisticated tool might flatten the 8-bit comparator into faster, non-iterative structure similar to the 74x682 MSI comparator (Figure 5-84 on page 421), and it might flatten our iterative cascading logic in Table 6-30 into two-level sum-of-products equations similar to the ones in the ABEL solution on page 483.

6.3.5 Mode-Dependent Comparator

For the next example, let us suppose we have a system in which we need to compare two 32-bit words under normal circumstances, but where we must sometimes ignore one or two low-order bits of the input words. The operating mode is specified by two mode-control bits, M1 and M0, as shown in Table 6-9 on page 484.

The desired functionality can be obtained very easily in VHDL using a `case` statement to select the behavior by mode, as shown in the program in Table 6-31. This is a perfectly good behavioral description that is also fully synthesizable. However, it has one major drawback in synthesis—it will, in all likelihood, cause the creation of three separate equality and magnitude comparators (32-, 31-, and 30-bit), one for each case in the `case` statement. The

Table 6-31 VHDL behavioral architecture of a 32-bit mode-dependent comparator.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vmodecmp is
    port ( M: in STD_LOGIC_VECTOR (1 downto 0);     -- mode
           A, B: in STD_LOGIC_VECTOR (31 downto 0); -- unsigned integers
           EQ, GT: out STD_LOGIC );                 -- comparison results
end Vmodecmp;

architecture Vmodecmp_arch of Vmodecmp is
begin
  process (M, A, B)
  begin
    case M is
      when "00" =>
        if A = B then EQ <= '1'; else EQ <= '0'; end if;
        if A > B then GT <= '1'; else GT <= '0'; end if;
      when "01" =>
        if A(31 downto 1) = B(31 downto 1) then EQ <= '1'; else EQ <= '0'; end if;
        if A(31 downto 1) > B(31 downto 1) then GT <= '1'; else GT <= '0'; end if;
      when "10" =>
        if A(31 downto 2) = B(31 downto 2) then EQ <= '1'; else EQ <= '0'; end if;
        if A(31 downto 2) > B(31 downto 2) then GT <= '1'; else GT <= '0'; end if;
      when others => EQ <= '0';  GT <= '0';
    end case;
  end process;
end Vmodecmp_arch;
```

individual comparators may or may not be fast, as discussed in the previous subsection, but we won't worry about speed for this example.

A more efficient alternative is to perform just one comparison for the 30 high-order bits of the inputs, and use additional logic that is dependent on mode to give a final result using the low-order bits as necessary. This approach is shown in Table 6-32. Two variables EQ30 and GT30 are used within the process to hold the results of the comparison of the 30 high-order bits. A case statement similar to the previous architecture's is then used to obtain the final results as a function of the mode. If desired, the speed of the 30-bit comparison can be optimized using the methods discussed in the preceding subsection.

6.3.6 Ones Counter

Several important algorithms include the step of counting the number of "1" bits in a data word. In fact, some microprocessor instruction sets have been extended recently to include ones counting as a basic instruction. In this example, let us suppose that we have a requirement to design a combinational circuit that counts ones in a 32-bit word as part of the arithmetic and logic unit of a microprocessor.

Table 6-32 More efficient architecture for a 32-bit mode-dependent comparator.

```
architecture Vmodecpe_arch of Vmodecmp is
begin
  process (M, A, B)
  variable EQ30, GT30: STD_LOGIC; -- 30-bit comparison results
  begin
    if A(31 downto 2) = B(31 downto 2) then EQ30 := '1'; else EQ30 := '0'; end if;
    if A(31 downto 2) > B(31 downto 2) then GT30 := '1'; else GT30 := '0'; end if;
    case M is
      when "00" =>
        if EQ30='1' and A(1 downto 0) = B(1 downto 0) then
          EQ <= '1'; else EQ <= '0'; end if;
        if GT30='1' or (EQ30='1' and A(1 downto 0) > B(1 downto 0)) then
          GT <= '1'; else GT <= '0'; end if;
      when "01" =>
        if EQ30='1' and A(1) = B(1) then EQ <= '1'; else EQ <= '0'; end if;
        if GT30='1' or (EQ30='1' and A(1) > B(1)) then
          GT <= '1'; else GT <= '0'; end if;
      when "10" =>  EQ <= EQ30;  GT <= GT30;
      when others => EQ <= '0';  GT <= '0';
    end case;
  end process;
end Vmodecpe_arch;
```

Ones counting can be described very easily by a behavioral VHDL program, as shown in Table 6-33. This program is fully synthesizable, but it may generate a very slow, inefficient realization with 32 5-bit adders in series.

Table 6-33 Behavioral VHDL program for a 32-bit ones counter.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vcnt1s is
    port ( D: in STD_LOGIC_VECTOR (31 downto 0);
           SUM: out STD_LOGIC_VECTOR (4 downto 0) );
end Vcnt1s;

architecture Vcnt1s_arch of Vcnt1s is
begin
  process (D)
  variable S: STD_LOGIC_VECTOR(4 downto 0);
  begin
    S := "00000";
    for i in 0 to 31 loop
      if D(i) = '1' then S := S + "00001"; end if;
    end loop;
    SUM <= S;
  end process;
end Vcnt1s_arch;
```

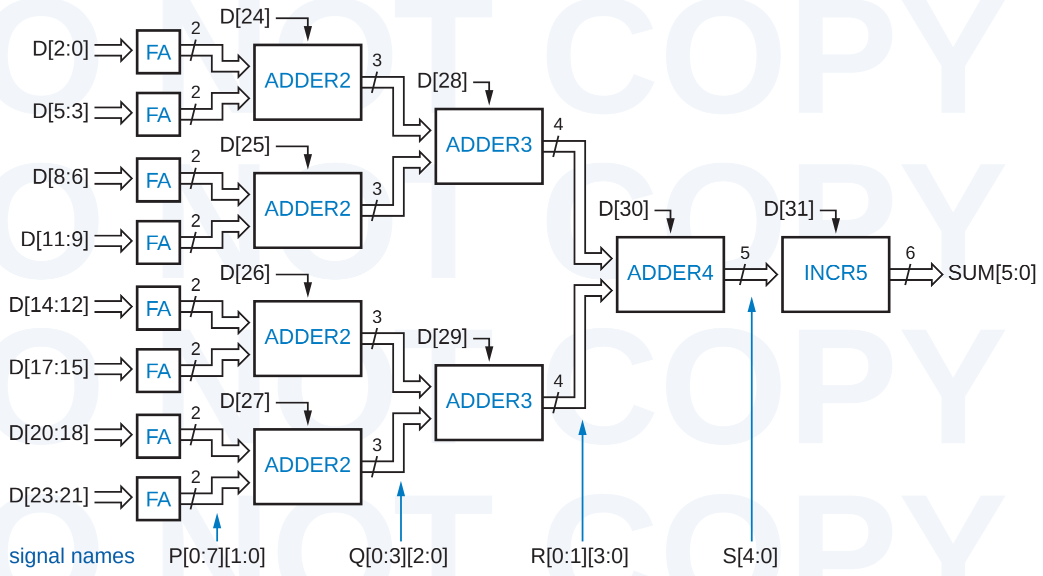


Figure 6-15 Structure of 32-bit ones counter.

To synthesize a more efficient realization of the ones counter, we must come up with an efficient structure and then write an architecture that describes it. Such a structure is the adder tree shown in Figure 6-15. A full adder (FA) adds three input bits to produce a 2-bit sum. Pairs of 2-bit numbers are added by 2-bit adders (ADDER2), each of which also has a carry input that can include add another 1-bit input to its sum. The resulting 3-bit sums are combined by 3-bit adders (ADDER3), and the final pair of 4-bit sums are combined in a 4-bit adder (ADDER4). By making use of the available carry inputs, this tree structure can combine 31 bits. A separate 5-bit incrementer is used at the end to handle the one remaining input bit.

The structure of Figure 6-15 can be created nicely by a structural VHDL architecture, as shown in Table 6-34. The program begins by declaring all of the components that will be used in the design, corresponding to the blocks in the figure.

The letter under each column of signals in Figure 6-15 corresponds to the name used for that signal in the program. Each of signals P, Q, and R is an array with one STD_LOGIC_VECTOR per connection in the corresponding column. The program defines a corresponding type for each of these, followed by the actual signal declaration.

The program in Table 6-34 makes good use of generate statements to create the multiple adder components on the left-hand side of the figure—eight FAs, four ADDER2s, and two ADDER3s. Finally, it instantiates one each of ADDER4 and INCR5.

Table 6-34 VHDL structural architecture for a 32-bit ones counter.

```
architecture Vcnt1str_arch of Vcnt1str is

component FA port ( A, B, CI: in  STD_LOGIC;
                    S, CO:    out STD_LOGIC );
end component;

component ADDER2 port ( A, B: in  STD_LOGIC_VECTOR(1 downto 0);
                        CI:   in  STD_LOGIC;
                        S:    out STD_LOGIC_VECTOR(2 downto 0) );
end component;

component ADDER3 port ( A, B: in  STD_LOGIC_VECTOR(2 downto 0);
                        CI:   in  STD_LOGIC;
                        S:    out STD_LOGIC_VECTOR(3 downto 0) );
end component;

component ADDER4 port ( A, B: in  STD_LOGIC_VECTOR(3 downto 0);
                        CI:   in  STD_LOGIC;
                        S:    out STD_LOGIC_VECTOR(4 downto 0) );
end component;

component INCR5 port ( A:  in  STD_LOGIC_VECTOR(4 downto 0);
                       CI: in  STD_LOGIC;
                       S:  out STD_LOGIC_VECTOR(5 downto 0) );
end component;

type Ptype is array (0 to 7) of STD_LOGIC_VECTOR(1 downto 0);
type Qtype is array (0 to 3) of STD_LOGIC_VECTOR(2 downto 0);
type Rtype is array (0 to 1) of STD_LOGIC_VECTOR(3 downto 0);
signal P: Ptype;  signal Q: Qtype;  signal R: Rtype;
signal S: STD_LOGIC_VECTOR(4 downto 0);

begin
  U1: for i in 0 to 7 generate
    U1C: FA port map (D(3*i), D(3*i+1), D(3*i+2), P(i)(0), P(i)(1));
  end generate;
  U2: for i in 0 to 3 generate
    U2C: ADDER2 port map (P(2*i), P(2*i+1), D(24+i), Q(i));
  end generate;
  U3: for i in 0 to 1 generate
    U3C: ADDER3 port map (Q(2*i), Q(2*i+1), D(28+i), R(i));
  end generate;
  U4: ADDER4 port map (R(0), R(1), D(30), S);
  U5: INCR5 port map (S, D(31), SUM);
end Vcnt1str_arch;
```

The definitions of the ones counter's individual component entities and architectures, from FA to INCR, can be made in separate structural or behavioral programs. For example, Table 6-35 is a structural program for FA. The rest of the components are left as exercises (6.20–6.22).

Table 6-35 Structural VHDL program for a full adder.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FA is
    port ( A, B, CI: in  STD_LOGIC;
           S, CO:    out STD_LOGIC );
end FA;

architecture FA_arch of FA is
begin
  S <= A xor B xor CI;
  CO <= (A and B) or (A and CI) or (B and CI);
end FA_arch;
```

6.3.7 Tic-Tac-Toe

Our last example is the design of a combinational circuit that picks a player's next move in the game of Tic-Tac-Toe. In case you're not familiar with the game, the rules are explained in the box on page 488. We'll repeat here our strategy for playing and winning the game:

1. Look for a row, column, or diagonal that has two of my marks (X or O, depending on which player I am) and one empty cell. If one exists, place my mark in the empty cell; I win!
2. Else, look for a row, column, or diagonal that has two of my opponent's marks and one empty cell. If one exists, place my mark in the empty cell to block a potential win by my opponent.
3. Else, pick a cell based on experience. For example, if the middle cell is open, it's usually a good bet to take it. Otherwise, the corner cells are good bets. Intelligent players can also notice and block a developing pattern by the opponent or "look ahead" to pick a good move.

To avoid confusion between "O" and "0" in our programs, we'll call the second player "Y". Now we can think about how to encode the inputs and outputs of the circuit. The inputs represent the current state of the playing grid. There are nine cells, and each cell has one of three possible states (empty, occupied by X, occupied by Y). The outputs represent the move to make, assuming that it is X's turn. There are only nine possible moves that a player can make, so the output can be encoded in just four bits.

There are several choices of how to code the state of one cell. Because the game is symmetric, we choose a symmetric encoding that can help us later:

00	Cell is empty.
10	Cell is occupied by X.
01	Cell is occupied by Y.

Table 6-36 VHDL package with definitions for the Tic-Tac-Toe project.

```
library IEEE;
use IEEE.std_logic_1164.all;

package TTTdefs is

type TTTgrid is array (1 to 3) of STD_LOGIC_VECTOR(1 to 3);
subtype TTTmove is STD_LOGIC_VECTOR (3 downto 0);

constant MOVE11: TTTmove := "1000";
constant MOVE12: TTTmove := "0100";
constant MOVE13: TTTmove := "0010";
constant MOVE21: TTTmove := "0001";
constant MOVE22: TTTmove := "1100";
constant MOVE23: TTTmove := "0111";
constant MOVE31: TTTmove := "1011";
constant MOVE32: TTTmove := "1101";
constant MOVE33: TTTmove := "1110";
constant NONE:   TTTmove := "0000";

end TTTdefs;
```

So, we can encode the 3 × 3 grid's state in 18 bits. Since VHDL supports arrays, it is useful to define an array type, `TTTgrid`, that contains elements corresponding to the cells in the grid. Since this type will be used throughout our Tic-Tac-Toe project, it is convenient to put this definition, along with several others that we'll come to, in a VHDL package, as shown in Table 6-36.

It would be natural to define `TTTgrid` as a two-dimensional array of `STD_LOGIC`, but not all VHDL tools support two-dimensional arrays. Instead, we define it as an array of 3-bit `STD_LOGIC_VECTOR`s, which is almost the same thing. To represent the Tic-Tac-Toe grid, we'll use two signals `X` and `Y` of this type, where an element of a variable is 1 if the like-named player has a mark in the corresponding cell. Figure 6-11 shows the correspondence between signal names and cells in the grid.

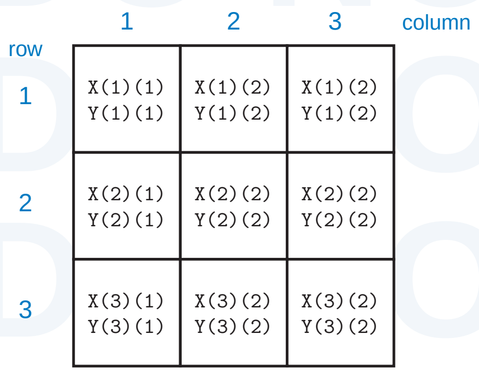


Figure 6-16 Tic-Tac-Toe grid and VHDL signal names.

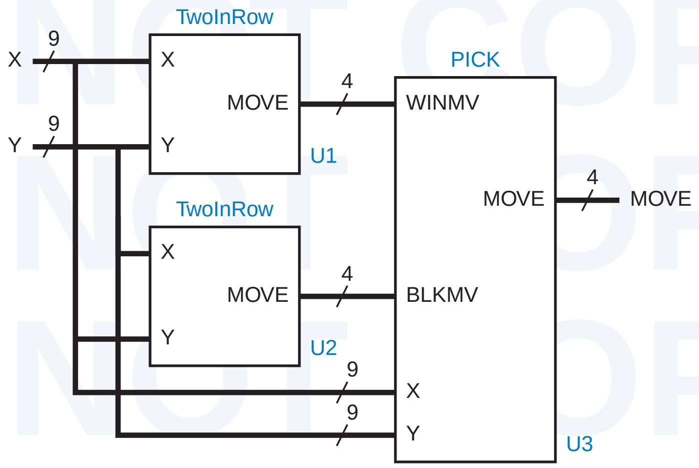


Figure 6-17
Entity partitioning for the Tic-Tac-Toe game.

The package in Table 6-36 also defines a 4-bit type `TTTmove` for encoded moves. A player has nine possible moves, and one more code is used for the case where no move is possible. This particular coding was chosen and used in the package for no other reason than that it's the same coding that was used in the ABEL version of this example in Section 6.2.7. By defining the coding in the package, we can easily change the definition later without having to change the entities that use it (for example, see Exercise 6.23).

Rather than try to design the Tic-Tac-Toe move-finding circuit as a single monolithic entity, it makes sense for us to try to partition it into smaller pieces. In fact, partitioning it along the lines of the three-step strategy at the beginning of this section seems like a good idea.

We note that steps 1 and 2 of our strategy are very similar; they differ only in reversing the roles of the player and the opponent. An entity that finds a winning move for me can also find a blocking move for my opponent. Looking at this characteristic from another point of view, an entity that finds a winning move for me can find a blocking move for me if the encodings for me and my opponent are swapped. Here's where our symmetric encoding pays off—we can swap players merely by swapping signals `X` and `Y`.

With this in mind, we can use two copies of the same entity, TwoInRow, to perform steps 1 and 2 as shown in Figure 6-17. Notice that signal X is connected to the top input of the first TwoInRow entity, but to the bottom input of the second. A third entity, `PICK`, picks a winning move if one is available from U1, else it picks a blocking move if available from U2, else it uses "experience" (step 3) to pick a move.

Table 6-37 is a structural VHDL program for the top-level entity, `GETMOVE`. In addition to the IEEE `std_logic_1164` package, it uses our `TTTdefs` package. Notice that the "`use`" clause for the `TTTdefs` packages specifies that it is stored in the "`work`" library, which is automatically created for our project.

Table 6-37
Top-level structural VHDL entity for picking a move in Tic-Tac-Toe.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity GETMOVE is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end GETMOVE;

architecture GETMOVE_arch of GETMOVE is

component TwoInRow port ( X, Y: in  TTTgrid;
                          MOVE: out STD_LOGIC_VECTOR(3 downto 0) );
end component;

component PICK port ( X, Y:          in  TTTgrid;
                      WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
                     MOVE:         out STD_LOGIC_VECTOR(3 downto 0) );
end component;

signal WIN, BLK: STD_LOGIC_VECTOR(3 downto 0);

begin
  U1: TwoInRow port map (X, Y, WIN);
  U2: TwoInRow port map (Y, X, BLK);
  U3: PICK port map (X, Y, WIN, BLK, MOVE);
end GETMOVE_arch;
```

The architecture in Table 6-37 declares and uses just two components, `TwoInRow` and `PICK`, which will be defined shortly. The only internal signals are `WIN` and `BLK`, which pass winning and blocking moves from the two instances of `TwoInRow` to `PICK`, as in Figure 6-17. The statement part of the architecture has just three statements to instantiate the three blocks in the figure.

Now comes the interesting part, the design of the individual entities in Figure 6-17. We'll start with `TwoInRow` since it accounts for two-thirds of the design. Its entity definition is very simple, as shown in Table 6-38. But there's plenty to discuss about its architecture, shown in Table 6-39.

Table 6-38
Declaration of TwoInRow entity.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity TwoInRow is
    port ( X, Y: in  TTTgrid;
           MOVE: out TTTmove );
end TwoInRow;
```

Table 6-39 Architecture of TwoInRow entity.

```
architecture TwoInRow_arch of TwoInRow is

function R(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                             -- Find 2-in-row with empty cell i,j
    result := TRUE;
    for jj in 1 to 3 loop
      if jj = j then result := result and X(i)(jj)='0' and Y(i)(jj)='0';
      else result := result and X(i)(jj)='1'; end if;
    end loop;
    return result;
  end R;

function C(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;
  begin                             -- Find 2-in-column with empty cell i,j
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(j)='0' and Y(ii)(j)='0';
      else result := result and X(ii)(j)='1'; end if;
    end loop;
    return result;
  end C;

function D(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;        -- Find 2-in-diagonal with empty cell i,j.
  begin                            -- This is for 11, 22, 33 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(ii)='0' and Y(ii)(ii)='0';
      else result := result and X(ii)(ii)='1'; end if;
    end loop;
    return result;
  end D;

function E(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  variable result: BOOLEAN;        -- Find 2-in-diagonal with empty cell i,j.
  begin                            -- This is for 13, 22, 31 diagonal.
    result := TRUE;
    for ii in 1 to 3 loop
      if ii = i then result := result and X(ii)(4-ii)='0' and Y(ii)(4-ii)='0';
      else result := result and X(ii)(4-ii)='1'; end if;
    end loop;
    return result;
  end E;
```

Table 6-39 (continued)

```
begin
  process (X, Y)
  variable G11, G12, G13, G21, G22, G23, G31, G32, G33: BOOLEAN;
  begin
    G11 := R(X,Y,1,1) or C(X,Y,1,1) or D(X,Y,1,1);
    G12 := R(X,Y,1,2) or C(X,Y,1,2);
    G13 := R(X,Y,1,3) or C(X,Y,1,3) or E(X,Y,1,3);
    G21 := R(X,Y,2,1) or C(X,Y,2,1);
    G22 := R(X,Y,2,2) or C(X,Y,2,2) or D(X,Y,2,2) or E(X,Y,2,2);
    G23 := R(X,Y,2,3) or C(X,Y,2,3);
    G31 := R(X,Y,3,1) or C(X,Y,3,1) or E(X,Y,3,1);
    G32 := R(X,Y,3,2) or C(X,Y,3,2);
    G33 := R(X,Y,3,3) or C(X,Y,3,3) or D(X,Y,3,3);
    if    G11 then MOVE <= MOVE11;
    elsif G12 then MOVE <= MOVE12;
    elsif G13 then MOVE <= MOVE13;
    elsif G21 then MOVE <= MOVE21;
    elsif G22 then MOVE <= MOVE22;
    elsif G23 then MOVE <= MOVE23;
    elsif G31 then MOVE <= MOVE31;
    elsif G32 then MOVE <= MOVE32;
    elsif G33 then MOVE <= MOVE33;
    else           MOVE <= NONE;
    end if;
  end process;
end TwoInRow_arch;
```

The architecture defines several functions, each of which determines whether there is a winning move (from X's point of view) in a particular cell i,j. A winning move exists if cell i,j is empty and the other two cells in the same row, column, or diagonal contain an X. Functions R and C look for winning moves in cell i,j's row and column, respectively. Functions D and E look in the two diagonals.

Within the architecture's single process, nine BOOLEAN variables G11–G33 are declared to indicate whether each of the cells has a winning move possible. Assignment statements at the beginning of the process set each variable to TRUE if there is such a move, calling and combining all of the appropriate functions for cell i,j.

The rest of the process is a deeply nested "if" statement that looks for a winning move in all possible cells. Although it typically results in slower synthesized logic nested "if" is required rather than some form of "case" statement, because multiple moves may be possible. If no winning move is possible, the value "NONE" is assigned.

EXPLICIT IMPURITY

In addition to a cell index `i,j`, the functions `R`, `C`, `D`, and `E` in Table 6-39 are passed the grid state `X` and `Y`. This is necessary because VHDL functions are by default *pure*, which means that signals and variables declared in the function's parents are *not* directly visible within the function. However, you can relax this restriction by explicitly declaring a function to be *impure* by placing the keyword `impure` before the keyword `function` in its definition.

The `PICK` entity combines the results of two `TwoInRow` entities according to the program in Table 6-40. First priority is given to a winning move, followed by a blocking move. Otherwise, function `MT` is called for each cell, starting with the middle and ending with the side cells, to find an available move. This completes the design of the Tic-Tac-Toe circuit.

Table 6-40 VHDL program to pick a winning or blocking Tic-Tac-Toe move, or else use "experience."

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.TTTdefs.all;

entity PICK is
  port ( X, Y:          in  TTTgrid;
         WINMV, BLKMV: in  STD_LOGIC_VECTOR(3 downto 0);
         MOVE:          out STD_LOGIC_VECTOR(3 downto 0) );
end PICK;

architecture PICK_arch of PICK is
function MT(X, Y: TTTgrid; i, j: INTEGER) return BOOLEAN is
  begin                          -- Determine if cell i,j is empty
    return X(i)(j)='0' and Y(i)(j)='0';
  end MT;
begin
  process (X, Y, WINMV, BLKMV)
  begin                                       -- If available, pick:
    if    WINMV /= NONE then MOVE <= WINMV;  -- winning move
    elsif BLKMV /= NONE then MOVE <= BLKMV;  -- else blocking move
    elsif MT(X,Y,2,2)   then MOVE <= MOVE22; -- else center cell
    elsif MT(X,Y,1,1)   then MOVE <= MOVE11; -- else corner cells
    elsif MT(X,Y,1,3)   then MOVE <= MOVE13;
    elsif MT(X,Y,3,1)   then MOVE <= MOVE31;
    elsif MT(X,Y,3,3)   then MOVE <= MOVE33;
    elsif MT(X,Y,1,2)   then MOVE <= MOVE12; -- else side cells
    elsif MT(X,Y,2,1)   then MOVE <= MOVE21;
    elsif MT(X,Y,2,3)   then MOVE <= MOVE23;
    elsif MT(X,Y,3,2)   then MOVE <= MOVE32;
    else                     MOVE <= NONE;   -- else grid is full
    end if;
  end process;
end PICK_arch;
```

Exercises

6.1 Explain how the 16-bit barrel shifter of Section 6.1.1 can be realized with a combination of 74x157s and 74x151s. How does this approach compare with the others in delay and parts count?

6.2 Show how the 16-bit barrel shifter of Section 6.1.1 can be realized in eight identical GAL22V10s.

6.3 Find a coding of the shift amounts (`S[3:0]`) and modes (`C[2:0]`) in the barrel shifter of Table 6-3 that further reduces the total number of product terms used by the design.

6.4 Make changes to the dual priority encoder program of Table 6-6 to further reduce the number of product terms required. State whether your changes increase the delay of the circuit when realized in a GAL22V10. Can you reduce the product terms enough to fit the design into a GAL16V8?

6.5 Here's an exercise where you can use your brain, like the author had to when figuring out the equation for the SUM0 output in Table 6-12. Do each of the SUM1–SUM3 outputs require more terms or fewer terms than SUM0?

6.6 Complete the design of the ABEL and PLD based ones counting circuit that was started in Section 6.2.6. Use 22V10 or smaller PLDs, and try to minimize the total number of PLDs required. State the total delay of your design in terms of the worst-case number of PLD delays in a signal path from input to output.

6.7 Find another code for the Tic-Tac-Toe moves in Table 6-13 that has the same rotation properties as the original code. That is, it should be possible to compensate for a 180° rotation of the grid using just inverters and wire rearrangement. Determine whether the TWOINHAF equations will still fit in a single 22V10 using the new code.

6.8 Using a simulator, demonstrate a sequence of moves in which the PICK2 PLD in Table 6-16 will lose a Tic-Tac-Toe game, even if X goes first.

6.9 Modify the program in Table 6-16 to give the program a better chance of winning, or at least not losing. Can your new program still lose?

6.10 Modify the both the "other logic" in Figure 6-13 and the program in Table 6-16 to give the program a better chance of winning, or at least not losing. Can your new program still lose?

6.11 Write the VHDL functions for `Vror`, `Vsll`, `Vsrl`, `Vsla`, and `Vsra` in Table 6-17 using the `ror`, `sll`, `srl`, `sla`, and `sra` operations as defined in Table 6-3.

6.12 The iterative-circuit version of `fixup` in Table 6-20 has a worst-case delay path of 15 OR gates from the first decoded value of `i` (14) to the `FSEL(0)` signal. Figure out a trick that cuts this delay path almost in half with no cost (or negative cost) in gates. How can this trick be extended further to save gates or gate inputs?

6.13 Rewrite the `barrel16` entity definition in Table 6-17 and the architecture in Table 6-22 so that a single direction-control bit is made explicitly available to the architecture.

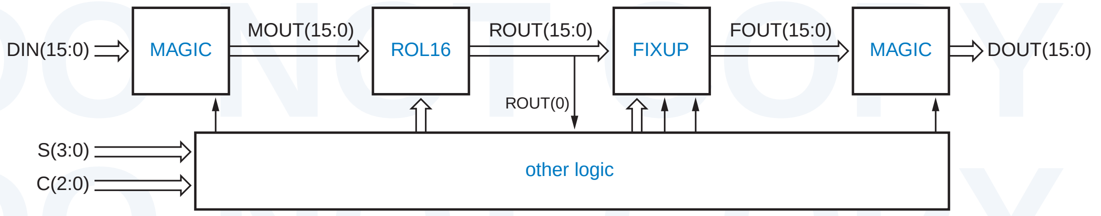


Figure X6.14

6.14 Rewrite the `barrel16` architecture definition in Table 6-22 to use the approach shown in Figure X6.14.

6.15 Write a semi-behavioral or structural version of the `fpencr_arch` architecture of Table 6-25 that generates only one adder in synthesis, and that does not generate multiple 10-bit comparators for the nested "`if`" statement.

6.16 Repeat Exercise 6.15, including a structural definition of an efficient rounding circuit that performs the `round` function. Your circuit should require significantly fewer gates than a 4-bit adder.

6.17 Redesign the VHDL dual priority encoder of Section 6.3.3 to get better, known performance, as suggested in the last paragraph of the section.

6.18 Write a structural VHDL architecture for a 64-bit comparator that is similar to Table 6-30 except that it builds up the comparison result serially from least to most significant stage.

6.19 What significant change occurs in the synthesis of the VHDL program in Table 6-31 if we change the statements in the "`when others`" case to "`null`"?

6.20 Write behavioral VHDL programs for the "ADDERx" components used in Table 6-34.

6.21 Write a structural VHDL programs for the "ADDERx" components in Table 6-34. Use a generic definition so that the same entity can be instantiated for ADDER2, ADDER3, and ADDER5, and show what changes must be made in Table 6-34 to do this.

6.22 Write a structural VHDL program for the "INCR5" component in Table 6-34.

6.23 Using an available VHDL synthesis tool, synthesize the Tic-Tac-Toe design of Section 6.3.7, fit it into an available FPGA, and determine how many internal resources it uses. Then try to reduce the resource requirements by specifying a different encoding of the moves in the `TTTdefs` package.

6.24 The Tic-Tac-Toe program in Section 6.3.7 eventually loses against an intelligent opponent if applied to the grid state shown in Figure X6.24. Use an available VHDL simulator to prove that this is true. Then modify the `PICK` entity to win in this and similar situations and verify your design using the simulator.

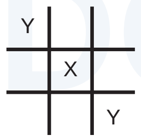
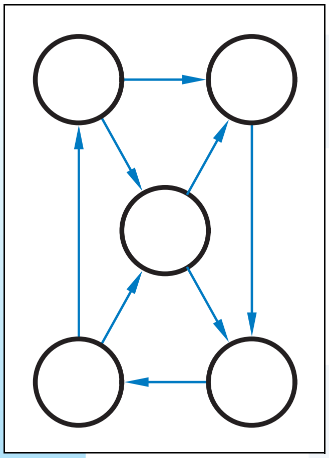


Figure X6.24



chapter 7

Sequential Logic Design Principles

Logic circuits are classified into two types, “combinational” and “sequential.” A *combinational* logic circuit is one whose outputs depend only on its current inputs. The rotary channel selector knob on an old-fashioned TV is like a combinational circuit—its “output” selects a channel based only on its current “input”—the position of the knob.

A *sequential* logic circuit is one whose outputs depend not only on its current inputs, but also on the past sequence of inputs, possibly arbitrarily far back in time. The circuit controlled by the channel-up and channel-down pushbuttons on a TV or VCR is a sequential circuit—the channel selection depends on the past sequence of up/down pushes, at least since when you started viewing 10 hours before, and perhaps as far back as when you first plugged the device into the wall.

So it is inconvenient, and often impossible, to describe the behavior of a sequential circuit by means of a table that lists outputs as a function of the input sequence that has been received up until the current time. To know where you’re going next, you need to know where you are now. With the TV channel selector, it is impossible to determine what channel is currently selected by looking only at the preceding sequence of presses on the up and down pushbuttons, whether we look at the preceding 10 presses or the preceding 1,000. More information, the current “state” of the channel selector, is needed. Probably the best definition of “state” that I’ve seen appeared in Herbert Hellerman's book on *Digital Computer System Principles* (McGraw-Hill, 1967):

state
state variable

> The *state* of a sequential circuit is a collection of *state variables* whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior.

In the channel-selector example, the current channel number is the current state. Inside the TV, this state might be stored as seven binary state variables representing a decimal number between 0 and 127. Given the current state (channel number), we can always predict the next state as a function of the inputs (presses of the up/down pushbuttons). In this example, one highly visible output of the sequential circuit is an encoding of the state itself—the channel-number display. Other outputs, internal to the TV, may be combinational functions of the state alone (e.g., VHF/UHF/cable tuner selection) or of both state and input (e.g., turning off the TV if the current state is 0 and the "down" button is pressed).

State variables need not have direct physical significance, and there are often many ways to choose them to describe a particular sequential circuit. For example, in the TV channel selector, the state might be stored as three BCD digits or 12 bits, with many of the bit combinations (4,096 possible) going unused.

In a digital logic circuit, state variables are binary values, corresponding to certain logic signals in the circuit, as we'll see in later sections. A circuit with *n* binary state variables has 2^n possible states. As large as it might be, 2^n is always finite, never infinite, so sequential circuits are sometimes called *finite-state machines*.

finite-state machine

The state changes of most sequential circuits occur at times specified by a free-running *clock* signal. Figure 7-1 gives timing diagrams and nomenclature for typical clock signals. By convention, a clock signal is active high if state changes occur at the clock's rising edge or when the clock is HIGH, and active low in the complementary case. The *clock period* is the time between successive transitions in the same direction, and the *clock frequency* is the reciprocal of the period. The first edge or pulse in a clock period or sometimes the period itself is called a *clock tick*. The *duty cycle* is the percentage of time that the clock signal

clock
clock period
clock frequency
clock tick
duty cycle

NON-FINITE-STATE MACHINES

A group of mathematicians recently proposed a non-finite-state machine, but they're still busy listing its states. . . . Sorry, that's just a joke. There *are* mathematical models for infinite-state machines, such as Turing machines. They typically contain a small finite-state-machine control unit, and an infinite amount of auxiliary memory, such as an endless tape.

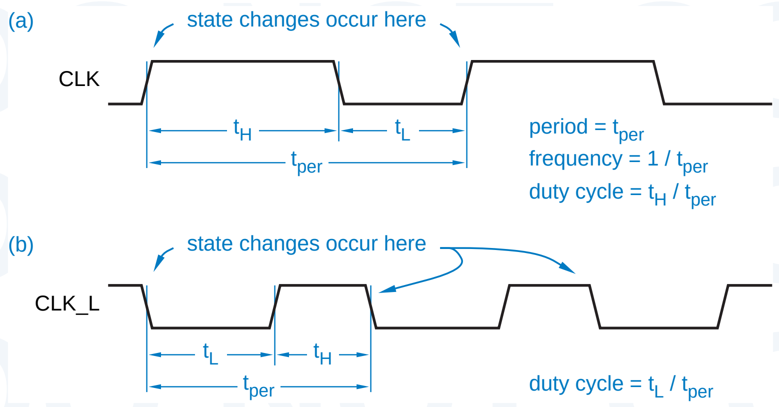


Figure 7-1 Clock signals: (a) active high; (b) active low.

is at its asserted level. Typical digital systems, from digital watches to supercomputers, use a quartz-crystal oscillator to generate a free-running clock signal. Clock frequencies might range from 32.768 kHz (for a watch) to 500 MHz (for a CMOS RISC microprocessor with a cycle time of 2 ns); "typical" systems using TTL and CMOS parts have clock frequencies in the 5–150 MHz range.

In this chapter we'll discuss two types of sequential circuits that account for the majority of practical discrete designs. A *feedback sequential circuit* uses ordinary gates and feedback loops to obtain memory in a logic circuit, thereby creating sequential-circuit building blocks such as latches and flip-flops that are used in higher-level designs. A *clocked synchronous state machine* uses these building blocks, in particular edge-triggered D flip-flops, to create circuits whose inputs are examined and whose outputs change in accordance with a controlling clock signal. There are other sequential circuit types, such as general fundamental mode, multiple pulse mode, and multiphase circuits, which are sometimes useful in high-performance systems and VLSI, and are discussed in advanced texts.

feedback sequential circuit

clocked synchronous state machine

7.1 Bistable Elements

The simplest sequential circuit consists of a pair of inverters forming a feedback loop, as shown in Figure 7-2. It has *no* inputs and two outputs, Q and Q_L.

7.1.1 Digital Analysis

The circuit of Figure 7-2 is often called a *bistable*, since a strictly digital analysis shows that it has two stable states. If Q is HIGH, then the bottom inverter has a HIGH input and a LOW output, which forces the top inverter's output HIGH as we assumed in the first place. But if Q is LOW, then the bottom inverter has a LOW input and a HIGH output, which forces Q LOW, another stable situation. We could use a single state variable, the state of signal Q, to describe the state of the circuit; there are two possible states, Q = 0 and Q = 1.

bistable

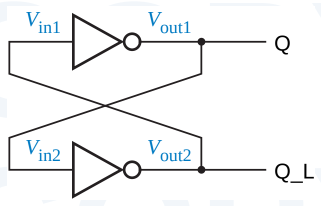


Figure 7-2
A pair of inverters forming a bistable element.

The bistable element is so simple that it has no inputs and therefore no way of controlling or changing its state. When power is first applied to the circuit, it randomly comes up in one state or the other and stays there forever. Still, it serves our illustrative purposes very well, and we *will* actually show a couple of applications for it in Sections 8.2.3 and 8.2.4.

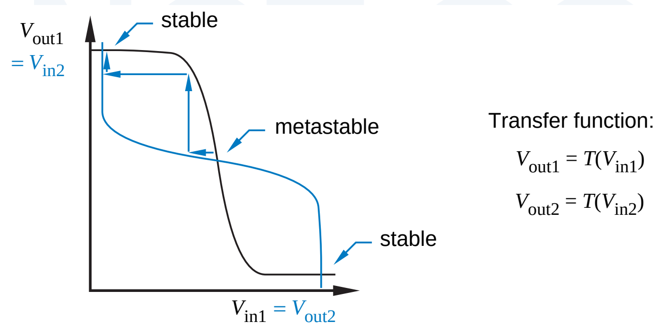
7.1.2 Analog Analysis

The analysis of the bistable has more to reveal if we consider its operation from an analog point of view. The dark line in Figure 7-3 shows the steady-state (DC) transfer function T for a single inverter; the output voltage is a function of input voltage, $V_{out} = T(V_{in})$. With two inverters connected in a feedback loop as in Figure 7-2, we know that $V_{in1} = V_{out2}$ and $V_{in2} = V_{out1}$; therefore, we can plot the transfer functions for both inverters on the same graph with an appropriate labeling of the axes. Thus, the dark line is the transfer function for the top inverter in Figure 7-2, and the colored line is the transfer function for the bottom one.

Considering only the steady-state behavior of the bistable's feedback loop, and not dynamic effects, the loop is in equilibrium if the input and output voltages of both inverters are constant DC values consistent with the loop connection and the inverters' DC transfer function. That is, we must have

$$\begin{aligned} V_{in1} &= V_{out2} \\ &= T(V_{in2}) \\ &= T(V_{out1}) \\ &= T(T(V_{in1})) \end{aligned}$$

Figure 7-3
Transfer functions for inverters in a bistable feedback loop.



Likewise, we must have

$$V_{\text{in2}} = T(T(V_{\text{in2}}))$$

We can find these equilibrium points graphically from Figure 7-3; they are the points at which the two transfer curves meet. Surprisingly, we find that there are not two but *three* equilibrium points. Two of them, labeled *stable*, correspond to the two states that our "strictly digital" analysis identified earlier, with Q either 0 (LOW) or 1 (HIGH).

stable

The third equilibrium point, labeled *metastable*, occurs with V_{out1} and V_{out2} about halfway between a valid logic 1 voltage and a valid logic 0 voltage; so Q and Q_L are not valid logic signals at this point. Yet the loop equations are satisfied; if we can get the circuit to operate at the metastable point, it could theoretically stay there indefinitely.

metastable

7.1.3 Metastable Behavior

Closer analysis of the situation at the metastable point shows that it is aptly named. It is not truly stable, because random noise will tend to drive a circuit that is operating at the metastable point toward one of the stable operating points as we'll now demonstrate.

Suppose the bistable is operating precisely at the metastable point in Figure 7-3. Now let us assume that a small amount of circuit noise reduces V_{in1} by a tiny amount. This tiny change causes V_{out1} to *increase* by a small amount. But since V_{out1} produces V_{in2}, we can follow the first horizontal arrow from near the metastable point to the second transfer characteristic, which now demands a lower voltage for V_{out2}, which is V_{in1}. Now we're back where we started, except we have a much larger change in voltage at V_{in1} than the original noise produced, and the operating point is still changing. This "regenerative" process continues until we reach the stable operating point at the upper left-hand corner of Figure 7-3. However, if we perform a "noise" analysis for either of the stable operating points, we find that feedback brings the circuit back toward the stable operating point, rather than away from it.

Metastable behavior of a bistable can be compared to the behavior of a ball dropped onto a hill, as shown in Figure 7-4. If we drop a ball from overhead, it will probably roll down immediately to one side of the hill or the other. But if it lands right at the top, it may precariously sit there for a while before random

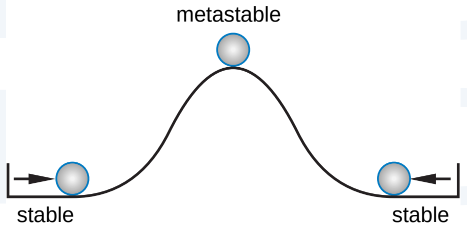


Figure 7-4
Ball and hill analogy for metastable behavior.

forces (wind, rodents, earthquakes) start it rolling down the hill. Like the ball at the top of the hill, the bistable may stay in the metastable state for an unpredictable length of time before nondeterministically settling into one stable state or the other.

If the *simplest* sequential circuit is susceptible to metastable behavior, you can be sure that *all* sequential circuits are susceptible. And this behavior is not something that only occurs at power-up.

Returning to the ball-and-hill analogy, consider what happens if we try to kick the ball from one side of the hill to the other. Apply a strong force (Arnold Schwarzenegger), and the ball goes right over the top and lands in a stable resting place on the other side. Apply a weak force (Mr. Rogers), and the ball falls back to its original starting place. But apply a wishy-washy force (Charlie Brown), and the ball goes to the top of the hill, teeters, and eventually falls back to one side or the other.

This behavior is completely analogous to what happens to flip-flops under marginal triggering conditions. For example, we'll soon study S-R flip-flops, where a pulse on the S input forces the flip-flop from the 0 state to the 1 state. A minimum pulse width is specified for the S input. Apply a pulse of this width or longer, and the flip-flop immediately goes to the 1 state. Apply a very short pulse, and the flip-flop stays in the 0 state. Apply a pulse just under the minimum width, and the flip-flop may go into the metastable state. Once the flip-flop is in the metastable state, its operation depends on "the shape of its hill." Flip-flops built from high-gain, fast technologies tend to come out of metastability faster than ones built from low-performance technologies.

We'll say more about metastability in the next section in connection with specific flip-flop types, and in Section 8.9 with respect to synchronous design methodology and synchronizer failure.

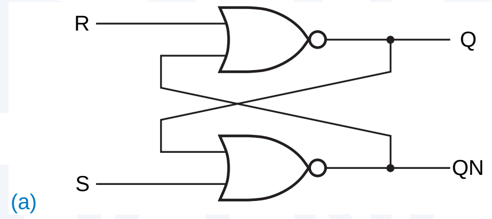
7.2 Latches and Flip-Flops

Latches and flip-flops are the basic building blocks of most sequential circuits. Typical digital systems use latches and flip-flops that are prepackaged, functionally specified devices in a standard integrated circuit. In ASIC design environments, latches and flip-flops are typically predefined cells specified by the ASIC vendor. However, within a standard IC or an ASIC, each latch or flip-flop cell is typically designed as a feedback sequential circuit using individual logic gates and feedback loops. We'll study these discrete designs for two reasons—to understand the behavior of the prepackaged elements better, and to gain the capability of building a latch or flip-flop "from scratch" as is required occasionally in digital-design practice and often in digital-design exams.

flip-flop

All digital designers use the name *flip-flop* for a sequential device that normally samples its inputs and changes its outputs only at times determined by a clocking signal. On the other hand, most digital designers use the name *latch* for

latch



(b)

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

Figure 7-5 S-R latch: (a) circuit design using NOR gates; (b) function table.

a sequential device that watches all of its inputs continuously and changes its outputs at any time, independent of a clocking signal. We follow this standard convention in this text. However, some textbooks and digital designers may (incorrectly) use the name "flip-flop" for a device that we call a "latch."

In any case, because the functional behaviors of latches and flip-flops are quite different, it is important for the logic designer to know which type is being used in a design, either from the device's part number (e.g., 74x374 vs. 74x373) or other contextual information. We discuss the most commonly used types of latches and flip-flops in the following subsections.

7.2.1 S-R Latch

S-R latch

An *S-R (set-reset) latch* based on NOR gates is shown in Figure 7-5(a). The circuit has two inputs, S and R, and two outputs, labeled Q and QN, where QN is normally the complement of Q. Signal QN is sometimes labeled $\overline{Q}$ or Q_L.

If S and R are both 0, the circuit behaves like the bistable element—we have a feedback loop that retains one of two logic states, Q = 0 or Q = 1. As shown in Figure 7-5(b), either S or R may be asserted to force the feedback loop to a desired state. S *sets* or *presets* the Q output to 1; R *resets* or *clears* the Q output to 0. After the S or R input is negated, the latch remains in the state that it was forced into. Figure 7-6(a) shows the functional behavior of an S-R latch for a typical sequence of inputs. Colored arrows indicate causality, that is, which input transitions cause which output transitions.

set
preset
reset
clear

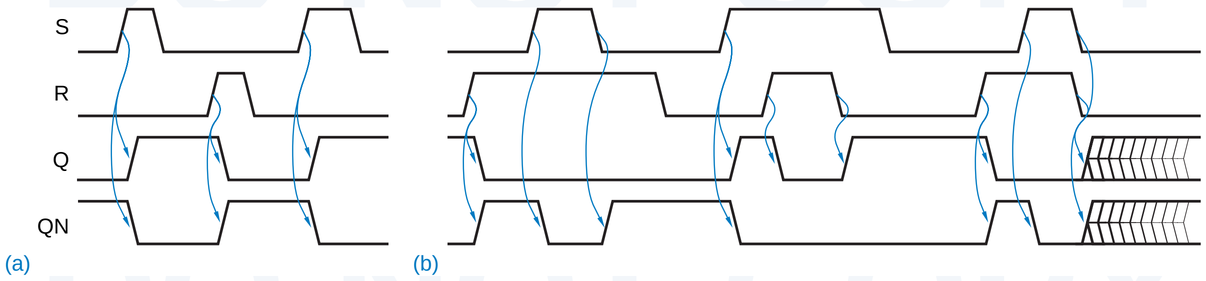


Figure 7-6 Typical operation of an S-R latch: (a) "normal" inputs; (b) S and R asserted simultaneously.

Q VERSUS QN

In most applications of an S-R latch, the QN (a.k.a. $\overline{\mathrm{Q}}$) output is always the complement of the Q output. However, the $\overline{\mathrm{Q}}$ name is not quite correct, because there is one case where this output is not the complement of Q. If both S and R are 1, as they are in several places in Figure 7-6(b), then both outputs are forced to 0. Once we negate either input, the outputs return to complementary operation as usual. However, if we negate both inputs simultaneously, the latch goes to an unpredictable next state, and it may in fact oscillate or enter the metastable state. Metastability may also occur if a 1 pulse that is too short is applied to S or R.

Three different logic symbols for the same S-R latch circuit are shown in Figure 7-7. The symbols differ in the treatment of the complemented output. Historically, the first symbol was used, showing the active-low or complemented signal name inside the function rectangle. However, in bubble-to-bubble logic design the second form of the symbol is preferred, showing an inversion bubble outside the function rectangle. The last form of the symbol is obviously wrong.

propagation delay

Figure 7-8 defines timing parameters for an S-R latch. The *propagation delay* is the time it takes for a transition on an input signal to produce a transition on an output signal. A given latch or flip-flop may have several different propagation delay specifications, one for each pair of input and output signals. Also, the propagation delay may be different depending on whether the output makes a LOW-to-HIGH or HIGH-to-LOW transition. With an S-R latch, a LOW-to-HIGH transition on S can cause a LOW-to-HIGH transition on Q, so a propagation delay $t_{\mathrm{pLH(SQ)}}$ occurs as shown in transition 1 in the figure. Similarly, a LOW-to-HIGH transition on R can cause a HIGH-to-LOW transition on Q, with propagation delay $t_{\mathrm{pHL(RQ)}}$ as shown in transition 2. Not shown in the figure are the corresponding transitions on QN, which would have propagation delays $t_{\mathrm{pHL(SQN)}}$ and $t_{\mathrm{pLH(RQN)}}$.

minimum pulse width

Minimum pulse width specifications are usually given for the S and R inputs. As shown in Figure 7-8, the latch may go into the metastable state and remain there for a random length of time if a pulse shorter than the minimum width $t_{\mathrm{pw(min)}}$ is applied to S or R. The latch can be deterministically brought out of the metastable state only by applying a pulse to S or R that meets or exceeds the minimum pulse width requirement.

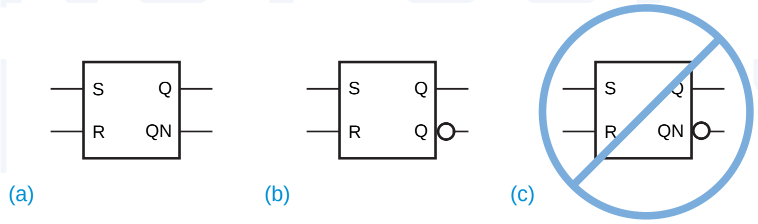


Figure 7-7 Symbols for an S-R latch: (a) without bubble; (b) preferred for bubble-to-bubble design; (c) incorrect because of double negation.

HOW CLOSE IS CLOSE?

As mentioned in the previous note, an S-R latch may go into the metastable state if S and R are negated simultaneously. Often, but not always, a commercial latch's specifications define "simultaneously" (e.g., S and R negated within 20 ns of each other). In any case, the minimum delay between negating S and R for them to be considered nonsimultaneous is closely related to the minimum pulse width specification. Both specifications are measures of how long it takes for the latch's feedback loop to stabilize during a change of state.

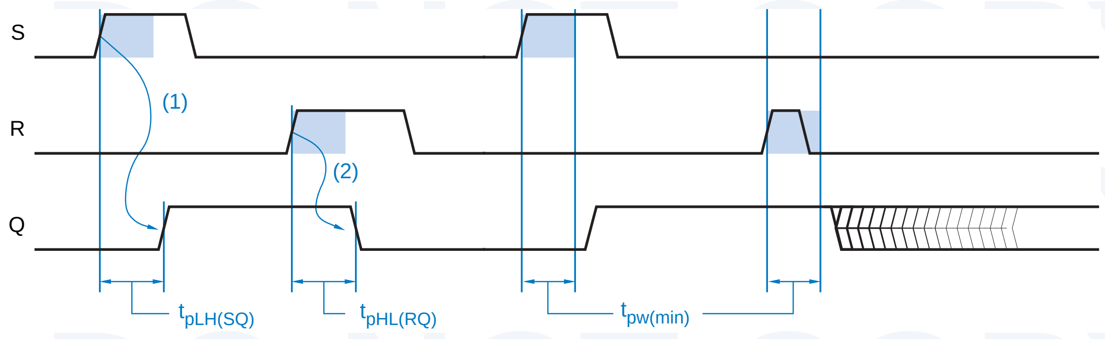


Figure 7-8 Timing parameters for an S-R latch.

7.2.2 $\overline{\text{S}}$-$\overline{\text{R}}$ Latch

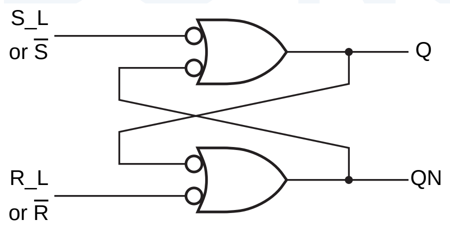
An *$\overline{S}$-$\overline{R}$ latch* (read "S-bar-R-bar latch") with active-low set and reset inputs may be built from NAND gates as shown in Figure 7-9(a). In TTL and CMOS logic families, $\overline{\text{S}}$-$\overline{\text{R}}$ latches are used much more often than S-R latches because NAND gates are preferred over NOR gates.

$\overline{S}$-$\overline{R}$ latch

As shown by the function table, Figure 7-9(b), operation of the $\overline{\text{S}}$-$\overline{\text{R}}$ latch is similar to that of the S-R, with two major differences. First, $\overline{\text{S}}$ and $\overline{\text{R}}$ are active low, so the latch remembers its previous state when $\overline{\text{S}} = \overline{\text{R}} = 1$; the active-low inputs are clearly indicated in the symbols in (c). Second, when both $\overline{\text{S}}$ and $\overline{\text{R}}$ are asserted simultaneously, both latch outputs go to 1, not 0 as in the S-R latch. Except for these differences, operation of the $\overline{\text{S}}$-$\overline{\text{R}}$ is the same as the S-R, including timing and metastability considerations.

Figure 7-9 $\overline{\text{S}}$-$\overline{\text{R}}$ latch: (a) circuit design using NAND gates; (b) function table; (c) logic symbol.

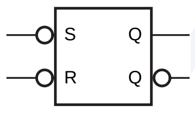
(a)



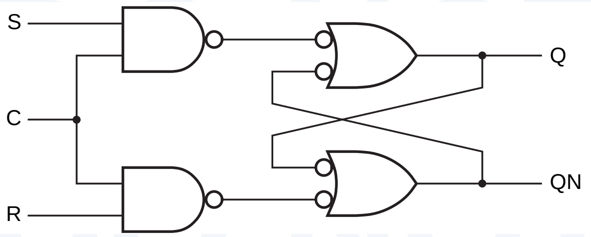
(b)

S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN

(c)



(a)



(b)

S	R	C	Q	QN
0	0	1	last Q	last QN
0	1	1	0	1
1	0	1	1	0
1	1	1	1	1
x	x	0	last Q	last QN

(c)

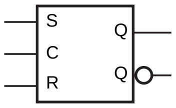


Figure 7-10 S-R latch with enable: (a) circuit using NAND gates; (b) function table; (c) logic symbol.

7.2.3 S-R Latch with Enable

S-R latch with enable

An S-R or $\overline{S}$-$\overline{R}$ latch is sensitive to its S and R inputs at all times. However, it may easily be modified to create a device that is sensitive to these inputs only when an enabling input C is asserted. Such an S-R *latch with enable* is shown in Figure 7-10. As shown by the function table, the circuit behaves like an S-R latch when C is 1, and retains its previous state when C is 0. The latch's behavior for a typical set of inputs is shown in Figure 7-11. If both S and R are 1 when C changes from 1 to 0, the circuit behaves like an S-R latch in which S and R are negated simultaneously—the next state is unpredictable and the output may become metastable.

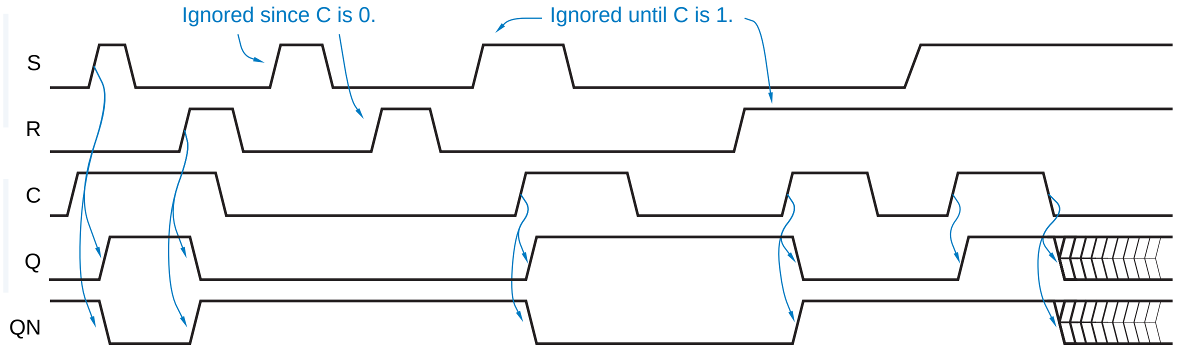
7.2.4 D Latch

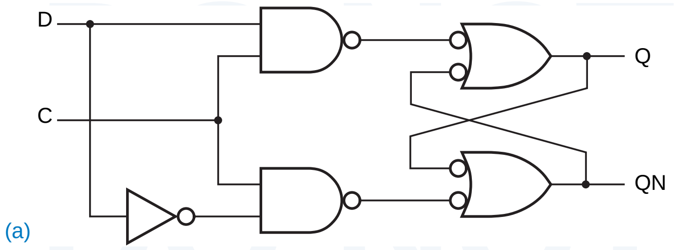
S-R latches are useful in control applications, where we often think in terms of setting a flag in response to some condition, and resetting it when conditions change; so we control the set and reset inputs somewhat independently. However, we often need latches simply to store bits of information—each bit of information is presented on a signal line, and we'd like to store it somewhere. A *D latch* may be used in such an application.

D latch

Figure 7-12 shows a D latch. Its logic diagram is recognizable as that of an S-R latch with enable, with an inverter added to generate S and R inputs from the

Figure 7-11 Typical operation of an S-R latch with enable.





C	D	Q	QN
1	0	0	1
1	1	1	0
0	x	last Q	last QN

(b)

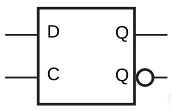


Figure 7-12 D latch: (a) circuit design using NAND gates; (b) function table; (c) logic symbol.

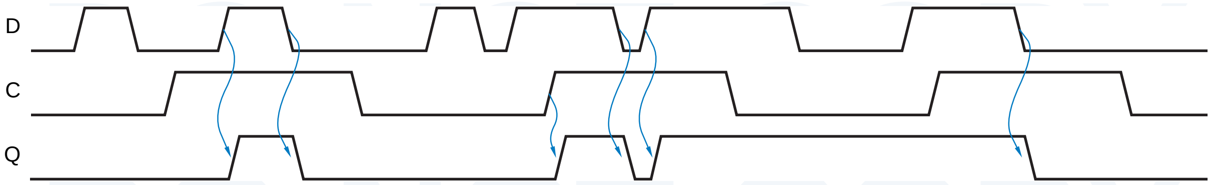


Figure 7-13 Functional behavior of a D latch for various inputs.

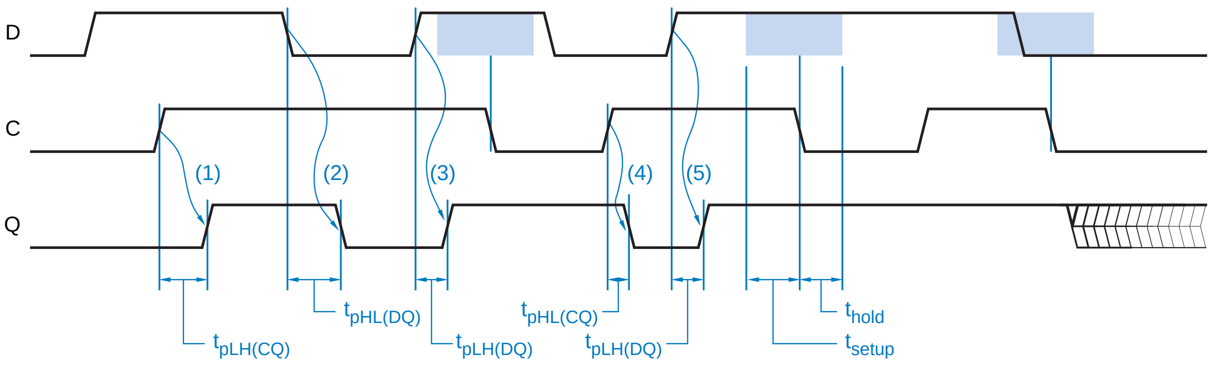
single D (data) input. This eliminates the troublesome situation in S-R latches, where S and R may be asserted simultaneously. The control input of a D latch, labeled C in (c), is sometimes named ENABLE, CLK, or G, and is active low in some D-latch designs.

An example of a D latch's functional behavior is given in Figure 7-13. When the C input is asserted, the Q output follows the D input. In this situation, the latch is said to be "open" and the path from D input to Q output is "transparent"; the circuit is often called a *transparent latch* for this reason. When the C input is negated, the latch "closes"; the Q output retains its last value and no longer changes in response to D, as long as C remains negated.

transparent latch

More detailed timing behavior of the D latch is shown in Figure 7-14. Four different delay parameters are shown for signals that propagate from the C or D input to the Q output. For example, at transitions 1 and 4 the latch is initially

Figure 7-14 Timing parameters for a D latch.



"closed" and the D input is the opposite of Q output, so that when C goes to 1 the latch "opens up" and the Q output changes after delay $t_{\text{pLH(CQ)}}$ or $t_{\text{pHL(CQ)}}$. At transitions 2 and 3 the C input is already 1 and the latch is already open, so that Q transparently follows the transitions on D with delay $t_{\text{pHL(DQ)}}$ and $t_{\text{pLH(DQ)}}$. Four more parameters specify the delay to the QN output, not shown.

Although the D latch eliminates the S = R = 1 problem of the S-R latch, it does not eliminate the metastability problem. As shown in Figure 7-14, there is a (shaded) window of time around the falling edge of C when the D input must not change. This window begins at time t_{setup} before the falling (latching) edge of C; t_{setup} is called the *setup time*. The window ends at time t_{hold} afterward; t_{hold} is called the *hold time*. If D changes at any time during the setup- and hold-time window, the output of the latch is unpredictable and may become metastable, as shown for the last latching edge in the figure.

setup time
hold time

7.2.5 Edge-Triggered D Flip-Flop

positive-edge-triggered D flip-flop

A *positive-edge-triggered D flip-flop* combines a pair of D latches, as shown in Figure 7-15, to create a circuit that samples its D input and changes its Q and QN outputs only at the rising edge of a controlling CLK signal. The first latch is called the *master*; it is open and follows the input when CLK is 0. When CLK goes to 1, the master latch is closed and its output is transferred to the second latch, called the *slave*. The slave latch is open all the while that CLK is 1, but changes only at the beginning of this interval, because the master is closed and unchanging during the rest of the interval.

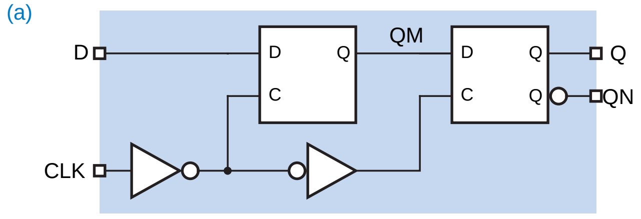
master
slave

dynamic-input indicator

The triangle on the D flip-flop's CLK input indicates edge-triggered behavior, and is called a *dynamic-input indicator*. Examples of the flip-flop's functional behavior for several input transitions are presented in Figure 7-16. The QM signal shown is the output of the master latch. Notice that QM changes only when CLK is 0. When CLK goes to 1, the current value of QM is transferred to Q, and QM is prevented from changing until CLK goes to 0 again.

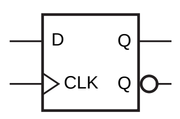
Figure 7-17 shows more detailed timing behavior for the D flip-flop. All propagation delays are measured from the rising edge of CLK, since that's the only event that causes an output change. Different delays may be specified for LOW-to-HIGH and HIGH-to-LOW output changes.

Figure 7-15 Positive-edge-triggered D flip-flop: (a) circuit design using D latches; (b) function table; (c) logic symbol.

(a)


(b)

D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

(c)


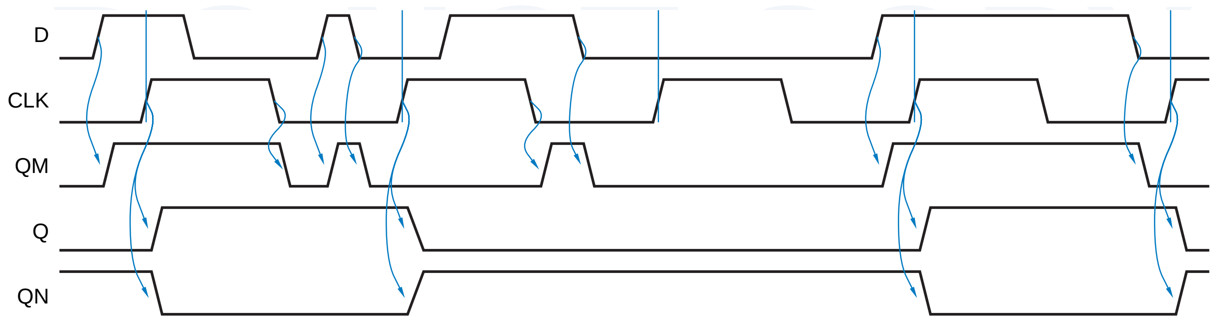


Figure 7-16 Functional behavior of a positive-edge-triggered D flip-flop.

Like a D latch, the edge-triggered D flip-flop has a setup and hold time window during which the D inputs must not change. This window occurs around the triggering edge of CLK, and is indicated by shaded color in Figure 7-17. If the setup and hold times are not met, the flip-flop output will usually go to a stable, though unpredictable, 0 or 1 state. In some cases, however, the output will oscillate or go to a metastable state halfway between 0 and 1, as shown at the second-to-last clock tick in the figure. If the flip-flop goes into the metastable state, it will return to a stable state on its own only after a probabilistic delay, as explained in \secref{metest}. It can also be forced into a stable state by applying another triggering clock edge with a D input that meets the setup- and hold-time requirements, as shown at the last clock tick in the figure.

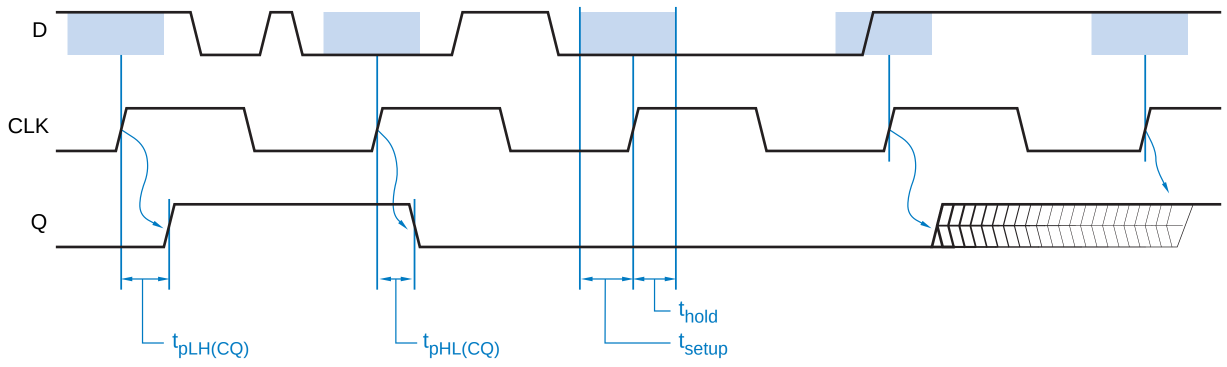
A *negative-edge-triggered* D *flip-flop* simply inverts the clock input, so that all the action takes place on the falling edge of CLK_L; by convention, a falling-edge trigger is considered to be active low. The flip-flop's function table and logic symbol are shown in Figure 7-18.

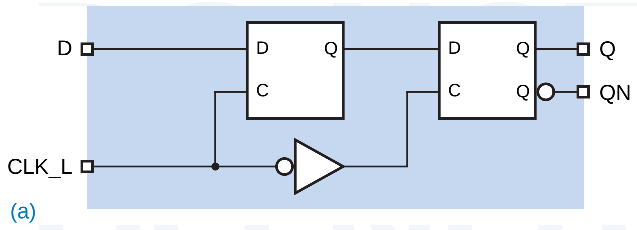
negative-edge-triggered D flip-flop

Some D flip-flops have *asynchronous inputs* that may be used to force the flip-flop to a particular state independent of the CLK and D inputs. These inputs, typically labeled PR (*preset*) and CLR (*clear*), behave like the set and reset

asynchronous inputs
preset
clear

Figure 7-17 Timing behavior of a positive-edge-triggered D flip-flop.





(b)

D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

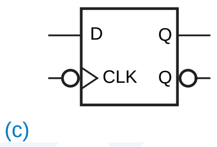


Figure 7-18 Negative-edge triggered D flip-flop: (a) circuit design using D latches; (b) function table; (c) logic symbol.

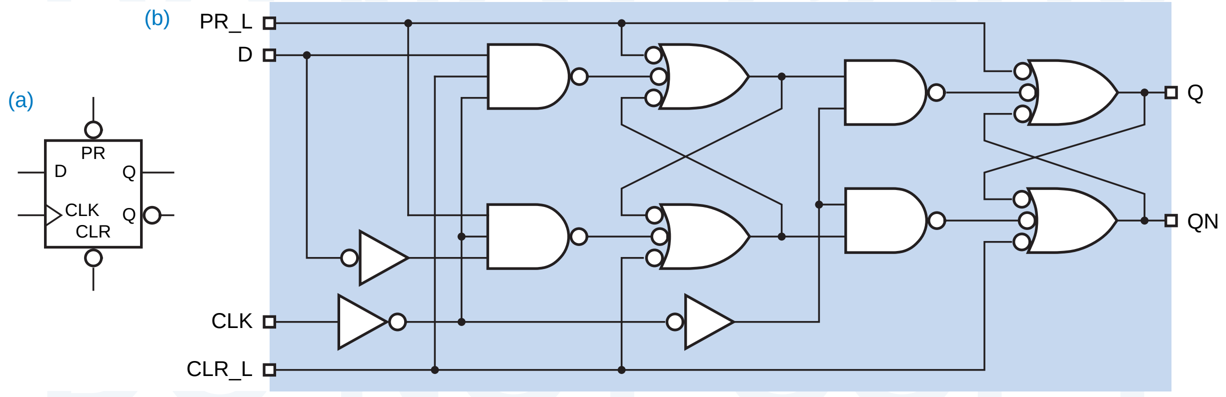
inputs on an SR latch. The logic symbol and NAND circuit for an edge-triggered D flip-flop with these inputs is shown in Figure 7-19. Although asynchronous inputs are used by some logic designers to perform tricky sequential functions, they are best reserved for initialization and testing purposes, to force a sequential circuit into a known starting state; more on this when we discuss synchronous design methodology in \secref{syncmethod}.

7.2.6 Edge-Triggered D Flip-Flop with Enable

enable input
clock-enable input

A commonly desired function in D flip-flops is the ability to hold the last value stored, rather than load a new value, at the clock edge. This is accomplished by adding an *enable input*, called EN or CE (*clock enable*). While the name "clock enable" is descriptive, the extra input's function is not obtained by controlling

Figure 7-19 Positive-edge-triggered D flip-flop with preset and clear: (a) logic symbol; (b) circuit design using NAND gates.



TIME FOR A COMMERCIAL

Commercial TTL positive-edge-triggered D flip-flops do not use the master-slave latch design of Figure 7-15 or Figure 7-19. Instead, flip-flops like the 74LS74 use the six-gate design of Figure 7-20, which is smaller and faster. We'll show how to formally analyze the next-state behavior of both designs in Section 7.9.

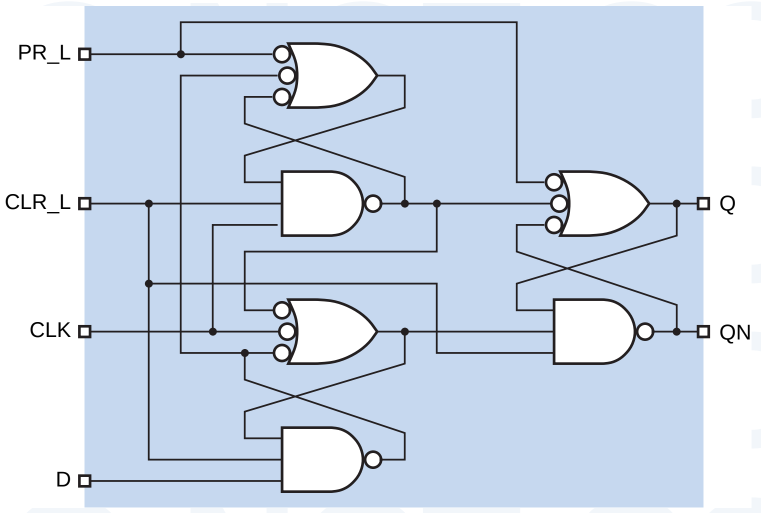


Figure 7-20 Commercial circuit for a positive-edge-triggered D flip-flop such as 74LS74.

the clock in any way whatsoever. Rather, as shown in Figure 7-21(a), a 2-input multiplexer controls the value applied to the internal flip-flop's D input. If EN is asserted, the external D input is selected; if EN is negated, the flip-flop's current output is used. The resulting function table is shown in (b). The flip-flop symbol is shown in (c); in some flip-flops, the enable input is active low, denoted by an inversion bubble on this input.

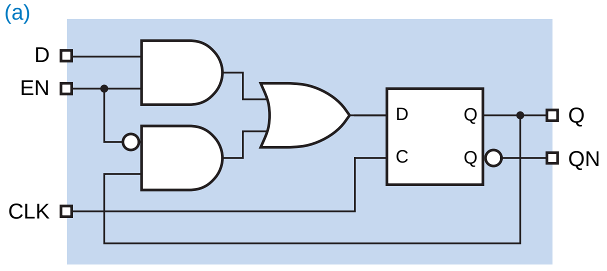
7.2.7 Scan Flip-Flop

scan capability

An important flip-flop function for ASIC testing is so-called *scan capability*. The idea is to be able to drive the flip-flop's D input with an alternate source of data during device testing. When all of the flip-flops are put into testing mode, a test pattern can be "scanned in" to the ASIC using the flip-flops' alternate data inputs. After the test pattern is loaded, the flip-flops are put back into "normal" mode, and all of the flip-flops are clocked normally. After one or more clock ticks, the flip-flops are put back into test mode, and the test results are "scanned out."

Figure 7-21 Positive-edge-triggered D flip-flop with enable: (a) circuit design; (b) function table; (c) logic symbol.

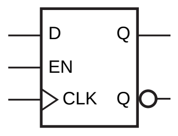
(a)

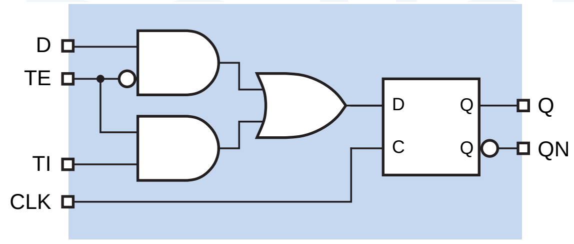


(b)

D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

(c)

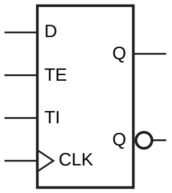




(a)

TE	TI	D	CLK	Q	QN
0	x	0	↑	0	1
0	x	1	↑	1	0
1	0	x	↑	0	1
1	1	x	↑	1	0
x	x	x	0	last Q	last QN
x	x	x	1	last Q	last QN

(b)



(c)

Figure 7-22 Positive-edge-triggered D flip-flop with scan: (a) circuit design; (b) function table; (c) logic symbol.

Figure 7-22(a) shows the design of a typical scan flip-flop. It is nothing more than a D flip-flop with a 2-input multiplexer on the D input. When the TE (*test enable*) input is negated, the circuit behaves like an ordinary D flip-flop. When TE is asserted, it takes its data from TI (*test input*) instead of from D. This functional behavior is shown in (b), and a symbol for the device is given in (c).

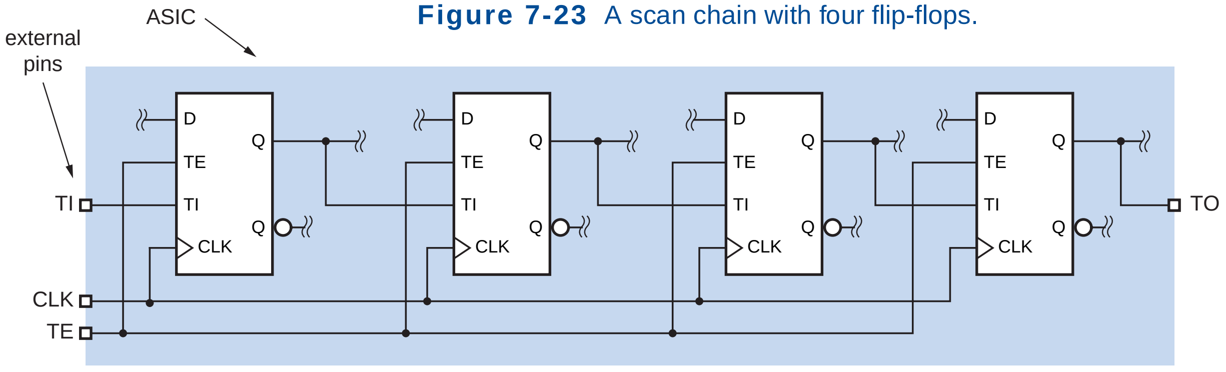
test-enable input, TE
test input, TI

scan chain

The extra inputs are used to connect all of an ASIC's flip-flops in a *scan chain* for testing purposes. Figure 7-23 is a simple example with four flip-flops in the scan chain. The TE inputs of all the flip-flops are connected to a global TE input, while each flip-flop's Q output is connected to another's TI input in serial (daisy-chain) fashion. The TI, TE, and TO (test output) connections are strictly for testing purposes; the additional logic connected to the D inputs and Q outputs needed to make the circuit do something useful are not shown.

To test the circuit, including the additional logic, the global TE input is asserted while *n* clock ticks occur and *n* test-vector bits are applied to the global TI input and are thereby scanned (shifted) into the *n* flip-flops; *n* equals 4 in Figure 7-23. Then TE is negated, and the circuit is allowed to run for one or more additional clock ticks. The new state of the circuit, represented by the new values in the *n* flip-flops, can be observed (scanned out) at TO by asserting TE while *n* more clock ticks occur. To make the testing process more efficient, another test vector can be scanned in while the previous result is being scanned out.

Figure 7-23 A scan chain with four flip-flops.



There are many different types of scan flip-flops, corresponding to different types of basic flip-flop functionality. For example, scan capability could be added to the D flip-flop with enable in Figure 7-21, by replacing its internal 2-input multiplexer with a 3-input one. At each clock tick, the flip-flop would load D, TI, or its current state depending on the values of EN and TE. Scan capability can also be added to other flip-flop types, such as J-K and T introduced later in this section.

*7.2.8 Master/Slave S-R Flip-Flop

We indicated earlier that S-R latches are useful in "control" applications, where we may have independent conditions for setting and resetting a control bit. If the control bit is supposed to be changed only at certain times with respect to a clock signal, then we need an S-R flip-flop that, like a D flip-flop, changes its outputs only on a certain edge of the clock signal. This subsection and the next two describe flip-flops that are useful for such applications.

master/slave S-R flip-flop

If we substitute S-R latches for the D latches in the negative-edge-triggered D flip-flop of Figure 7-18(a), we get a *master/slave S-R flip-flop*, shown in Figure 7-24. Like a D flip-flop, the S-R flip-flop changes its outputs only at the falling edge of a control signal C. However, the new output value depends on input values not just at the falling edge, but during the entire interval in which C is 1 prior to the falling edge. As shown in Figure 7-25, a short pulse on S any time during this interval can set the master latch; likewise, a pulse on R can reset it. The value transferred to the flip-flop output on the falling edge of C depends on whether the master latch was last set or cleared while C was 1.

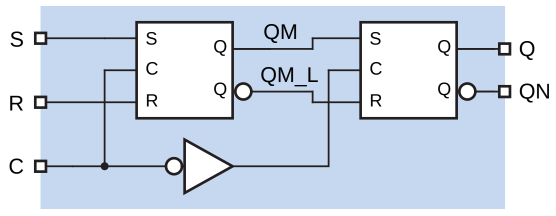
postponed-output indicator

pulse-triggered flip-flop

Shown in Figure 7-24(c), the logic symbol for the master/slave S-R flip-flop does not use a dynamic-input indicator, because the flip-flop is not truly edge triggered. It is more like a latch that follows its input during the entire interval that C is 1, but that changes its output to reflect the final latched value only when C goes to 0. In the symbol, a *postponed-output indicator* indicates that the output signal does not change until enable input C is negated. Flip-flops with this kind of behavior are sometimes called *pulse-triggered flip-flops*.

Figure 7-24 Master/slave S-R flip-flop: (a) circuit using S-R latches; (b) function table; (c) logic symbol.

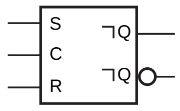
(a)



(b)

S	R	C	Q	QN
x	x	0	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	undef.	undef.

(c)



*Throughout this book, optional sections are marked with an asterisk.

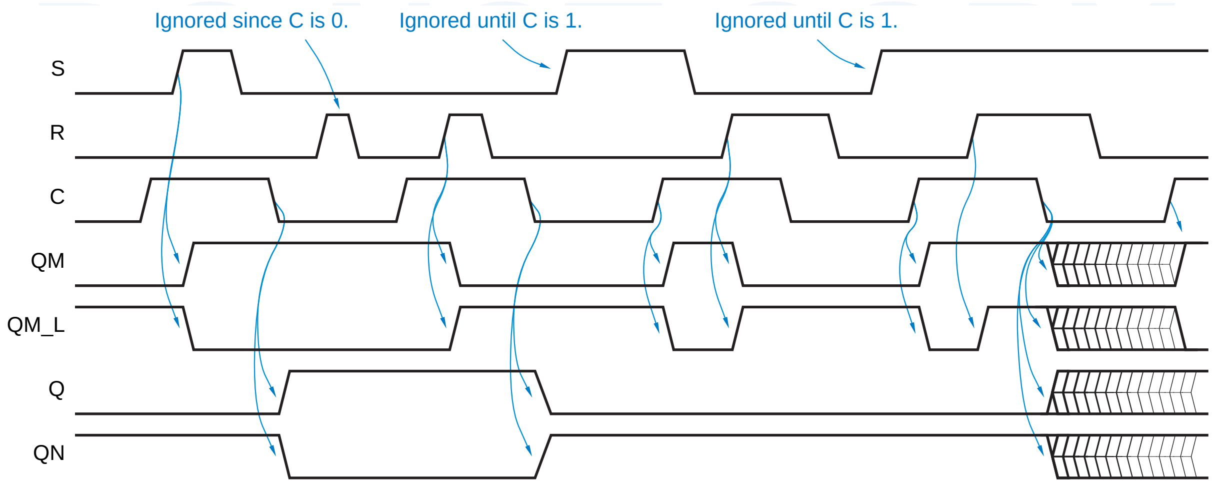


Figure 7-25 Internal and functional behavior of a master/slave S-R flip-flop.

The operation of the master/slave S-R flip-flop is unpredictable if both S and R are asserted at the falling edge of C. In this case, just before the falling edge, both the Q and QN outputs of the master latch are 1. When C goes to 0, the master latch's outputs change unpredictably and may even become metastable. At the same time, the slave latch opens up and propagates this garbage to the flip-flop output.

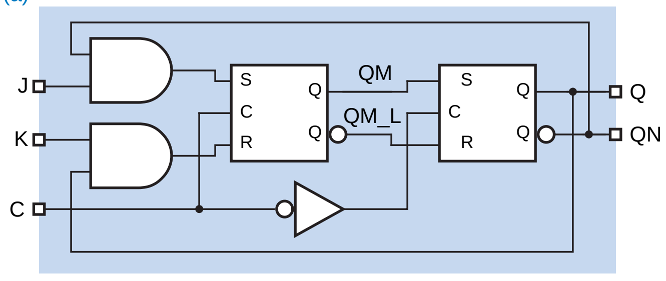
*7.2.9 Master/Slave J-K Flip-Flop

master/slave J-K flip-flop

The problem of what to do when S and R are asserted simultaneously is solved in a *master/slave J-K flip-flop*. The J and K inputs are analogous to S and R. However, as shown in Figure 7-26, asserting J asserts the master's S input only if the flip-flop's QN output is currently 1 (i.e., Q is 0), and asserting K asserts the master's R input only if Q is currently 1. Thus, if J and K are asserted simultaneously, the flip-flop goes to the opposite of its current state.

Figure 7-26 Master/slave J-K flip-flop: (a) circuit design using S-R latches; (b) function table; (c) logic symbol.

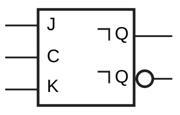
(a)



(b)

J	K	C	Q	QN
x	x	0	last Q	last QN
0	0	⎍	last Q	last QN
0	1	⎍	0	1
1	0	⎍	1	0
1	1	⎍	last QN	last Q

(c)



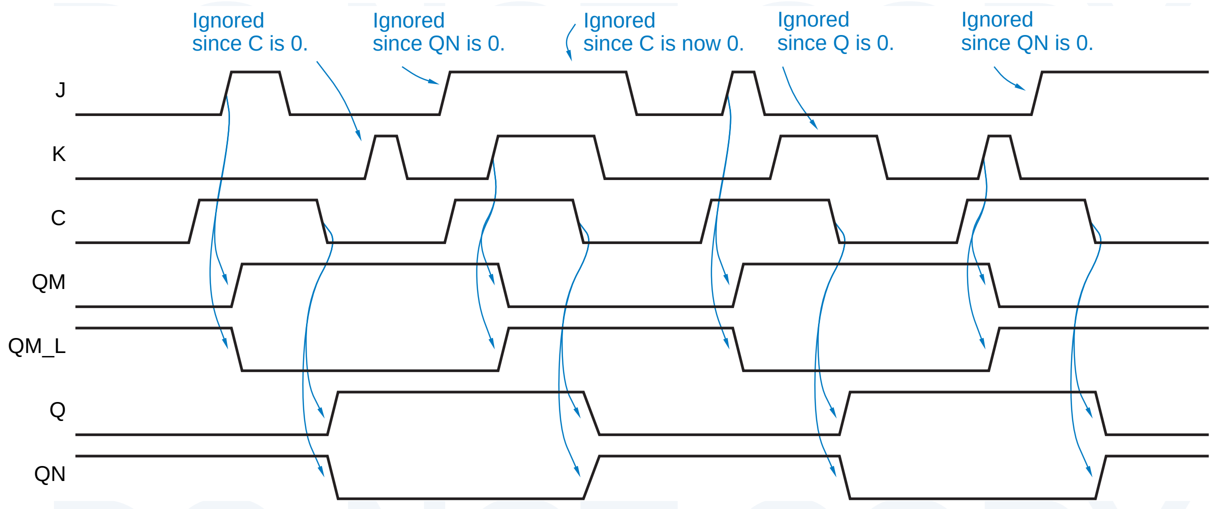


Figure 7-27 Internal and functional behavior of a master/slave J-K flip-flop.

Figure 7-27 shows the functional behavior of a J-K master/slave flip-flop for a typical set of inputs. Note that the J and K inputs need not be asserted at the end of the triggering pulse for the flip-flop output to change at that time. In fact, because of the gating on the master latch's S and R inputs, it is possible for the flip-flop output to change to 1 even though K and not J is asserted at the end of the triggering pulse. This behavior, known as *1s catching*, is illustrated in the second-to-last triggering pulse in the figure. An analogous behavior known as *0s catching* is illustrated in the last triggering pulse. Because of this behavior, the J and K inputs of a J-K master/slave flip-flop must be held valid during the entire interval that C is 1.

1s catching

0s catching

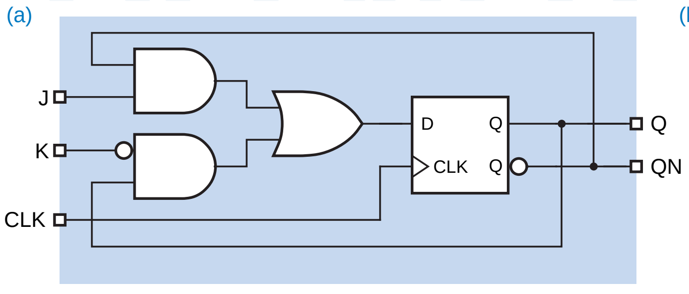
7.2.10 Edge-Triggered J-K Flip-Flop

The problem of 1s and 0s catching is solved in an *edge-triggered* J-K *flip-flop*, whose functional equivalent is shown in Figure 7-28. Using an edge-triggered D flip-flop internally, the edge-triggered J-K flip-flop samples its inputs at the

edge-triggered J-K flip-flop

Figure 7-28 Edge-triggered J-K flip-flop: (a) equivalent function using an edge-triggered D flip-flop; (b) function table; (c) logic symbol.

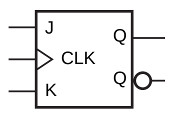
(a)



(b)

J	K	CLK	Q	QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN
0	0	↑	last Q	last QN
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	last QN	last Q

(c)



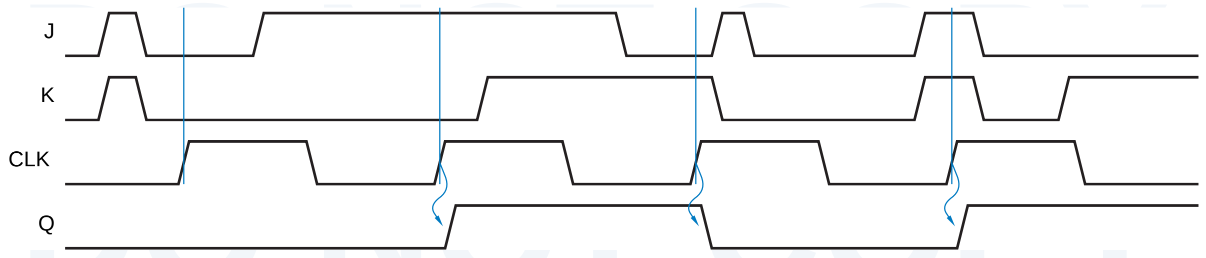


Figure 7-29 Functional behavior of a positive-edge-triggered J-K flip-flop.

rising edge of the clock and produces its next output according to the "characteristic equation" $Q^* = J \cdot Q' + K' \cdot Q$ (see Section 7.3.3).

Typical functional behavior of an edge-triggered J-K flip-flop is shown in Figure 7-29. Like the D input of an edge-triggered D flip-flop, the J and K inputs of a J-K flip-flop must meet published setup- and hold-time specifications with respect to the triggering clock edge for proper operation.

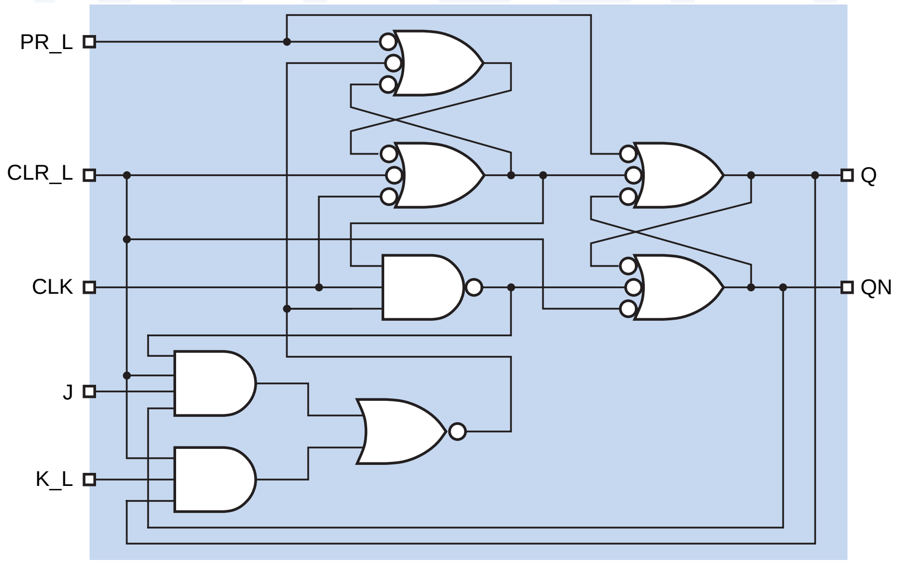
74x109

Because they eliminate the problems of 1s and 0s catching and of simultaneously asserting both control inputs, edge-triggered J-K flip-flops have largely obsoleted the older pulse-triggered types. The *74x109* is a TTL positive-edge-triggered J-$\overline{K}$ flip-flop with an active-low K input (named $\overline{K}$ or K_L).

ANOTHER COMMERCIAL (FLIP-FLOP, THAT IS)

The internal design of the 74LS109 is very similar to that of the 74LS74, which we showed in Figure 7-20. As shown in Figure 7-30, the '109 simply replaces the bottom-left gate of the '74, which realizes the characteristic equation $Q^* = D$, with an AND-OR structure that realizes the J-$\overline{K}$ characteristic equation, $Q^* = J \cdot Q' + K\_L \cdot Q$.

Figure 7-30 Internal logic diagram for the 74LS109 positive-edge-triggered J-$\overline{K}$ flip-flop.



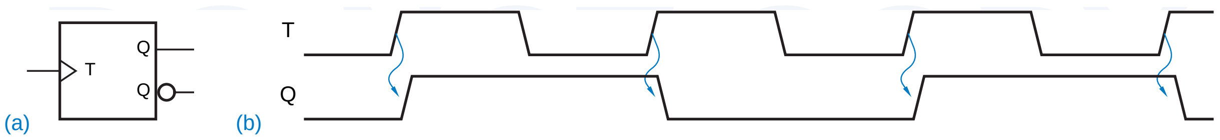


Figure 7-31 Positive-edge-triggered T flip-flop: (a) logic symbol; (b) functional behavior.

The most common application of J-K flip-flops is in clocked synchronous state machines. As we'll explain in Section 7.4.5, the next-state logic for J-K flip-flops is sometimes simpler than for D flip-flops. However, most state machines are still designed using D flip-flops because the design methodology is a bit simpler and because most sequential programmable logic devices contain D, not J-K, flip-flops. Therefore, we'll give most of our attention to D flip-flops.

7.2.11 T Flip-Flop

T flip-flop

A *T (toggle) flip-flop* changes state on every tick of the clock. Figure 7-31 shows the symbol and illustrates the behavior of a positive-edge-triggered T flip-flop. Notice that the signal on the flip-flop's Q output has precisely half the frequency of the T input. Figure 7-32 shows how to obtain a T flip-flop from a D or J-K flip-flop. T flip-flops are most often used in counters and frequency dividers, as we'll show in \secref{counters}.

T flip-flop with enable

In many applications of T flip-flops, the flip-flop need not be toggled on every clock tick. Such applications can use a *T flip-flop with enable*. As shown in Figure 7-33, the flip-flop changes state at the triggering edge of the clock only if the enable signal EN is asserted. Like the D, J, and K inputs on other edge-triggered flip-flops, the EN input must meet specified setup and hold times with respect to the triggering clock edge. The circuits of Figure 7-32 are easily modified to provide an EN input, as shown in Figure 7-34.

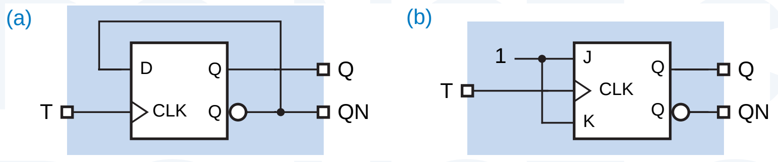
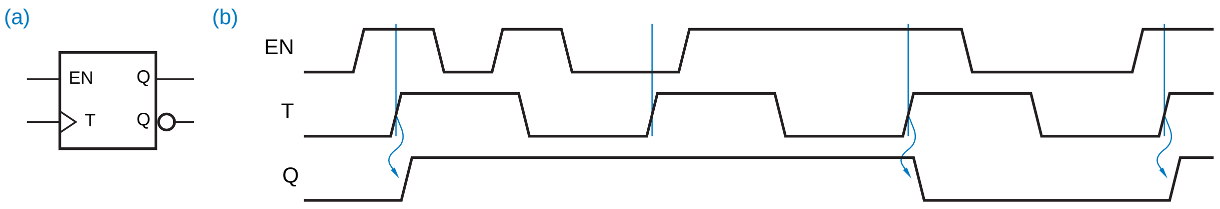


Figure 7-32 Possible circuit designs for a T flip-flop: (a) using a D flip-flop; (b) using a J-K flip-flop.

Figure 7-33 Positive-edge-triggered T flip-flop with enable: (a) logic symbol; (b) functional behavior.



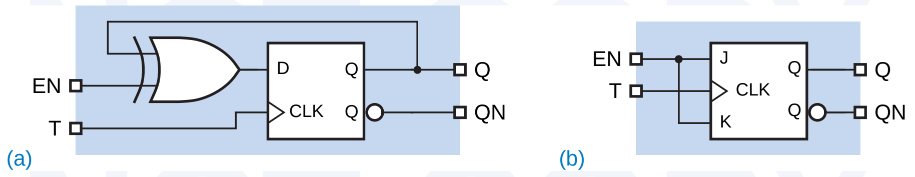


Figure 7-34 Possible circuits for a T flip-flop with enable: (a) using a D flip-flop; (b) using a J-K flip-flop.

7.3 Clocked Synchronous State-Machine Analysis

clocked synchronous state machine

Although latches and flip-flops, the basic building blocks of sequential circuits, are themselves feedback sequential circuits that can be formally analyzed, we'll first study the operation of *clocked synchronous state machines*, since they are the easiest to understand. "State machine" is a generic name given to these sequential circuits; "clocked" refers to the fact that their storage elements (flip-flops) employ a clock input; and "synchronous" means that all of the flip-flops use the same clock signal. Such a state machine changes state only when a triggering edge or "tick" occurs on the clock signal.

7.3.1 State-Machine Structure

state memory

tick

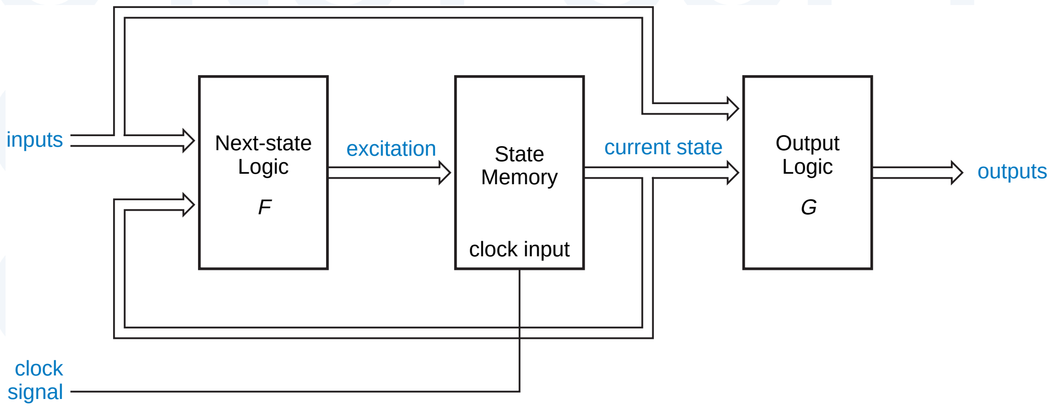
Figure 7-35 shows the general structure of a clocked synchronous state machine. The *state memory* is a set of *n* flip-flops that store the current state of the machine, and has 2^n distinct states. The flip-flops are all connected to a common clock signal that causes the flip-flops to change state at each *tick* of the clock. What constitutes a tick depends on the flip-flop type (edge triggered, pulse triggered, etc.). For the positive-edge-triggered D and J-K flip-flops considered in this section, a tick is the rising edge of the clock signal.

next-state logic

output logic

The next state of the state machine in Figure 7-35 is determined by the *next-state logic F* as a function of the current state and input. The *output logic G*

Figure 7-35 Clocked synchronous state-machine structure (Mealy machine).



determines the output as a function of the current state and input. Both F and G are strictly combinational logic circuits. We can write

$$\begin{aligned}\text{Next state} &= F(\text{current state, input}) \\ \text{Output} &= G(\text{current state, input})\end{aligned}$$

State machines may use positive-edge-triggered D flip-flops for their state memory, in which case a tick occurs at each rising edge of the clock signal. It is also possible for the state memory to use negative-edge-triggered D flip-flops, D latches, or J-K flip-flops. However, inasmuch as most state machines are designed nowadays using programmable logic devices with positive-edge-triggered D flip-flops, that's what we'll concentrate on.

7.3.2 Output Logic

A sequential circuit whose output depends on both state and input as shown in Figure 7-35 is called a *Mealy machine*. In some sequential circuits, the output depends on the state alone:

$$\text{Output} = G(\text{current state})$$

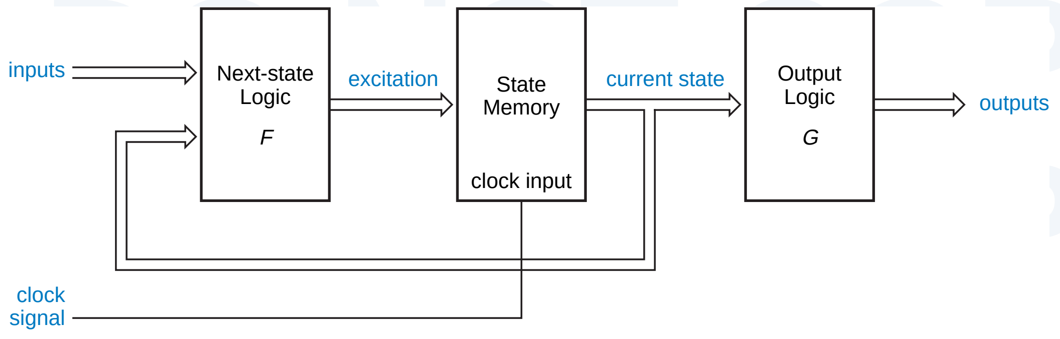
Such a circuit is called a *Moore machine*, and its general structure is shown in Figure 7-36.

Obviously, the only difference between the two state-machine models is in how outputs are generated. In practice, many state machines must be categorized as Mealy machines, because they have one or more *Mealy-type outputs* that depend on input as well as state. However, many of these same machines also have one or more *Moore-type outputs* that depend only on state.

In the design of high-speed circuits, it is often necessary to ensure that state-machine outputs are available as early as possible and do not change during each clock period. One way to get this behavior is to encode the state so that the state variables themselves serve as outputs. We call this an *output-coded state assignment*; it produces a Moore machine in which the output logic of Figure 7-36 is null, consisting of just wires.

output-coded state assignment

Figure 7-36 Clocked synchronous state-machine structure (Moore machine).



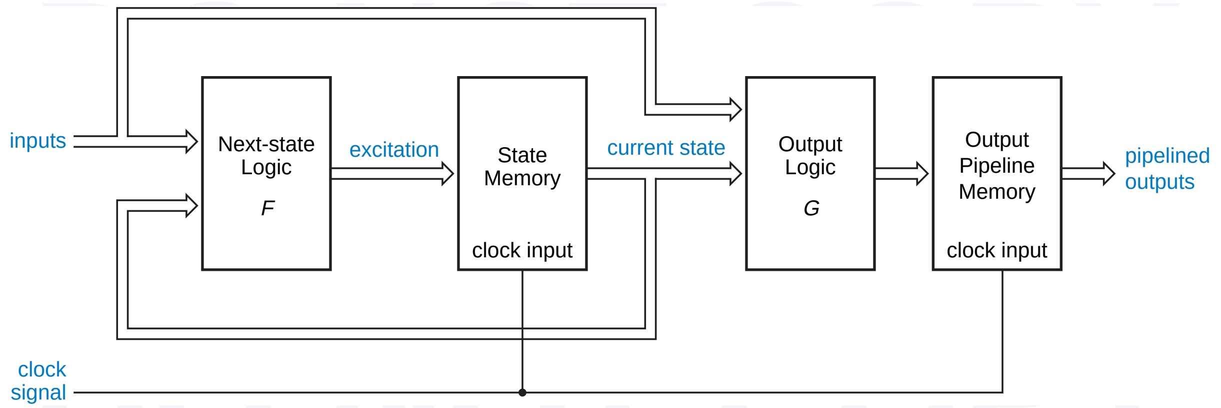


Figure 7-37 Mealy machine with pipelined outputs.

pipelined outputs

Another approach is to design the state machine so that the outputs during one clock period depend on the state and inputs during the *previous* clock period. We call these *pipelined outputs*, and they are obtained by attaching another stage of memory (flip-flops) to a Mealy machine's outputs as in Figure 7-37.

With appropriate circuit or drawing manipulations, you can map one state-machine model into another. For example, you could declare the flip-flops that produce pipelined outputs from a Mealy machine to be part of its state memory, and thereby obtain a Moore machine with an output-coded state assignment.

The exact classification of a state machine into one style or another is not so important. What's important is how you think about output structure and how it satisfies your overall design objectives, including timing and flexibility. For example, pipelined outputs are great for timing, but you can use them only in situations where you can figure out the desired next output value one clock period in advance. In any given application, you may use different styles for different output signals. For example, we'll see in Section 7.11.5 that different statements can be used to specify different output styles in ABEL.

7.3.3 Characteristic Equations

characteristic equation

The functional behavior of a latch or flip-flop can be described formally by a *characteristic equation* that specifies the flip-flop's next state as a function of its current state and inputs.

∗ suffix

The characteristic equations of the flip-flops in Section 7.2 are listed in Table 7-1. By convention, the ∗ suffix in Q∗ means "the next value of Q." Notice that the characteristic equation does not describe detailed timing behavior of the device (latching vs. edge-triggered, etc.), only the functional response to the control inputs. This simplified description is useful in the analysis of state machines, as we'll soon show.

Device Type	***Characteristic Equation***
S-R latch	$Q* = S + R' \cdot Q$
D latch	$Q* = D$
Edge-triggered D flip-flop	$Q* = D$
D flip-flop with enable	$Q* = EN \cdot D + EN' \cdot Q$
Master/slave S-R flip-flop	$Q* = S + R' \cdot Q$
Master/slave J-K flip-flop	$Q* = J \cdot Q' + K' \cdot Q$
Edge-triggered J-K flip-flop	$Q* = J \cdot Q' + K' \cdot Q$
T flip-flop	$Q* = Q'$
T flip-flop with enable	$Q* = EN \cdot Q' + EN' \cdot Q$

Table 7-1
Latch and flip-flop characteristic equations.

7.3.4 Analysis of State Machines with D Flip-Flops

Consider the formal definition of a state machine that we gave previously:

$$\text{Next state} = F(\text{current state, input})$$
$$\text{Output} = G(\text{current state, input})$$

Recalling our notion that "state" embodies all we need to know about the past history of the circuit, the first equation tells us that what we next need to know can be determined from what we currently know and the current input. The second equation tells us that the current output can be determined from the same information. The goal of sequential circuit analysis is to determine the next-state and output functions so that the behavior of a circuit can be predicted.

The analysis of a clocked synchronous state machine has three basic steps:

1. Determine the next-state and output functions *F* and *G*.
2. Use *F* and *G* to construct a *state/output table* that completely specifies the next state and output of the circuit for every possible combination of current state and input.

state/output table

3. (Optional) Draw a *state diagram* that presents the information from the previous step in graphical form.

state diagram

Figure 7-38 shows a simple state machine with two positive-edge-triggered D flip-flops. To determine the next-state function *F*, we must first consider the behavior of the state memory. At the rising edge of the clock signal, each D flip-flop samples its D input and transfers this value to its Q output; the characteristic equation of a D flip-flop is $Q* = D$. Therefore, to determine the next value of Q (i.e., Q*), we must first determine the current value of D.

In Figure 7-38 there are two D flip-flops, and we have named the signals on their outputs Q0 and Q1. These two outputs are the state variables; their value is

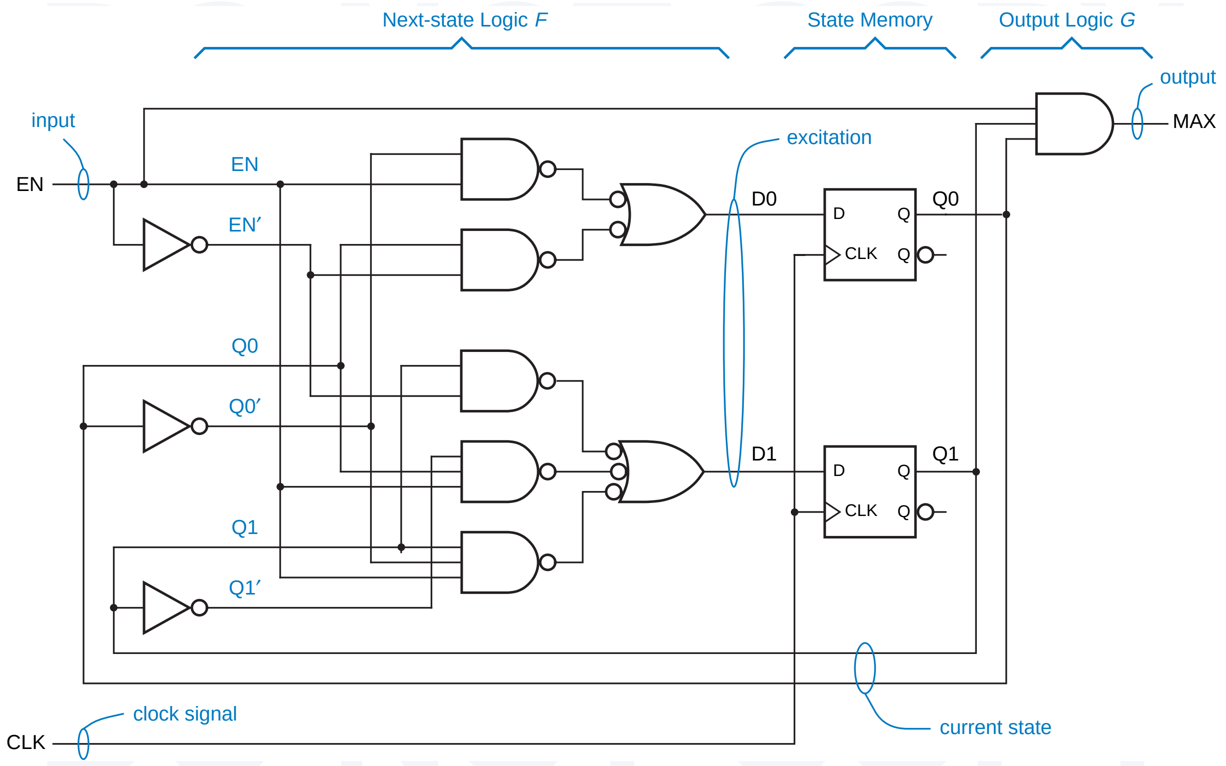


Figure 7-38 Clocked synchronous state machine using positive-edge-triggered D flip-flops.

excitation

excitation equation

the current state of the machine. We have named the signals on the corresponding D inputs D0 and D1. These signals provide the *excitation* for the D flip-flops at each clock tick. Logic equations that express the excitation signals as functions of the current state and input are called *excitation equations* and can be derived from the circuit diagram:

$$\begin{aligned} D0 &= Q0 \cdot EN' + Q0' \cdot EN \\ D1 &= Q1 \cdot EN' + Q1' \cdot Q0 \cdot EN + Q1 \cdot Q0' \cdot EN \end{aligned}$$

** suffix*

By convention, the next value of a state variable after a clock tick is denoted by appending a star to the state-variable name, for example, Q0∗ or Q1∗. Using the characteristic equation of D flip-flops, Q∗ = D, we can describe the next-state function of the example machine with equations for the next value of the state variables:

$$\begin{aligned} Q0* &= D0 \\ Q1* &= D1 \end{aligned}$$

Substituting the excitation equations for D0 and D1, we can write

$$\begin{aligned} Q0* &= Q0 \cdot EN' + Q0' \cdot EN \\ Q1* &= Q1 \cdot EN' + Q1' \cdot Q0 \cdot EN + Q1 \cdot Q0' \cdot EN \end{aligned}$$

Table 7-2
Transition, state, and state/output tables for the state machine in Figure 7-38.

(a)

	EN	
Q1 Q0	***0***	***1***
00	00	01
01	01	10
10	10	11
11	11	00
	Q1∗ Q0∗	

(b)

	EN	
S	***0***	***1***
A	A	B
B	B	C
C	C	D
D	D	A
	S∗	

(c)

	EN	
S	***0***	***1***
A	A, 0	B, 0
B	B, 0	C, 0
C	C, 0	D, 0
D	D, 0	A, 1
	S∗, MAX	

These equations, which express the next value of the state variables as a function of current state and input, are called *transition equations*.

transition equation

For each combination of current state and input value, the transition equations predict the next state. Each state is described by two bits, the current values of Q0 and Q1: (Q1 Q0) = 00, 01, 10, or 11. [The reason for “arbitrarily” picking the order (Q1 Q0) instead of (Q0 Q1) will become apparent shortly.] For each state, our example machine has just two possible input values, EN = 0 or EN = 1, so there are a total of 8 state/input combinations. (In general, a machine with *s* state bits and *i* inputs has 2^{s+i} state/input combinations.)

transition table

Table 7-2(a) shows a *transition table* that is created by evaluating the transition equations for every possible state/input combination. Traditionally, a transition table lists the states along the left and the input combinations along the top of the table, as shown in the example.

The function of our example machine is apparent from its transition table—it is a 2-bit binary counter with an enable input EN. When EN = 0 the machine maintains its current count, but when EN = 1 the count advances by 1 at each clock tick, rolling over to 00 when it reaches a maximum value of 11.

state names

If we wish, we may assign alphanumeric *state names* to each state. The simplest naming is 00 = A, 01 = B, 10 = C, and 11 = D. Substituting the state names for combinations of Q1 and Q0 (and Q1∗ and Q0∗) in Table 7-2(a) produces the *state table* in (b). Here “S” denotes the current state, and “S∗” denotes the next state of the machine. A state table is usually easier to understand than a transition table because in complex machines we can use state names that have meaning. However, a state table contains less information than a transition table because it does not indicate the binary values assumed by the state variables in each named state.

state table

Once a state table is produced, we have only the output logic of the machine left to analyze. In the example machine, there is only a single output signal, and it is a function of both current state and input (this is a Mealy machine). So we can write a single *output equation*:

output equation

$$\text{MAX} = \text{Q1} \cdot \text{Q0} \cdot \text{EN}$$

The output behavior predicted by this equation can be combined with the next-state information to produce a *state/output table* as shown in Table 7-2(c).

state/output table

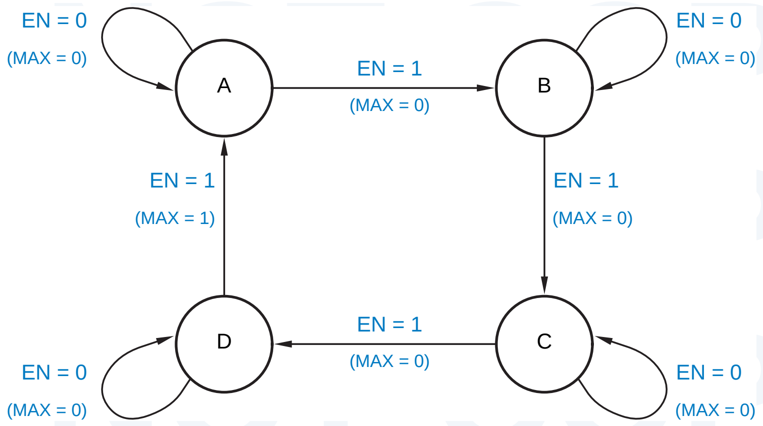


Figure 7-39
State diagram corresponding to the state machine of Table 7-2.

State/output tables for Moore machines are slightly simpler. For example, in the circuit of Figure 7-38 suppose we removed the EN signal from the AND gate that produces the MAX output, producing a Moore-type output MAXS. Then MAXS is a function of the state only, and the state/output table can list MAXS in a single column, independent of the input values. This is shown in Table 7-3.

state diagram
node
directed arc

A *state diagram* presents the information from the state/output table in a graphical format. It has one circle (or *node*) for each state, and an arrow (or *directed arc*) for each transition. Figure 7-39 shows the state diagram for our example state machine. The letter inside each circle is a state name. Each arrow leaving a given state points to the next state for a given input combination; it also shows the output value produced in the given state for that input combination.

The state diagram for a Moore machine can be somewhat simpler. In this case, the output values can be shown inside each state circle, since they are func-

Table 7-3
State/output table for a Moore machine.

	EN		
S	***0***	***1***	***MAXS***
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S*		

A CLARIFICATION The state-diagram notation for output values in Mealy machines is a little misleading. You should remember that the listed output value is produced continuously when the machine is in the indicated state and has the indicated input, not just during the transition to the next state.

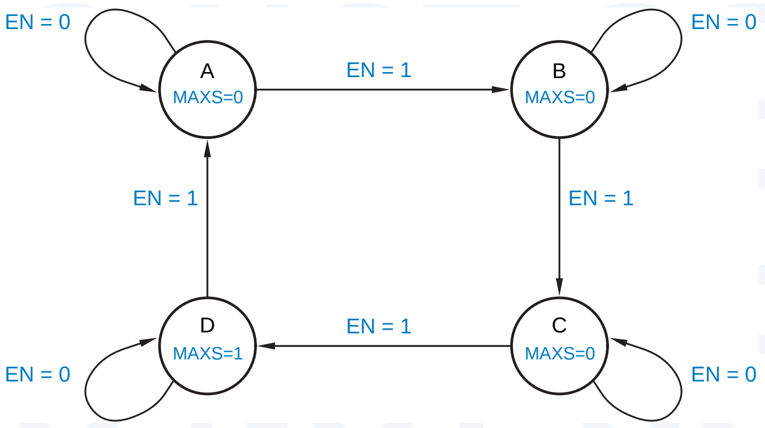
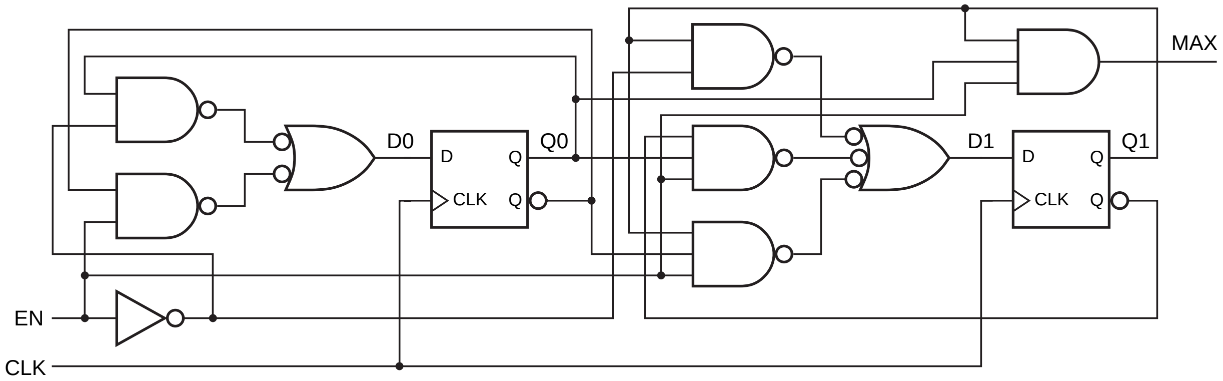


Figure 7-40 State diagram corresponding to the state machine of Table 7-3.

tions of state only. The state diagram for a Moore machine using this convention is shown in Figure 7-40.

The original logic diagram of our example state machine, Figure 7-38, was laid out to match our conceptual model of a Mealy machine. However, nothing requires us to group the next-state logic, state memory, and output logic in this way. Figure 7-41 shows another logic diagram for the same state machine. To analyze this circuit, the designer (or analyzer, in this case) can still extract the required information from the diagram as drawn. The only circuit difference in

Figure 7-41 Redrawn logic diagram for a clocked synchronous state machine.



LITTLE ARROWS, LITTLE ARROWS EVERYWHERE

Since there is only one input in our example machine, there are only two possible input combinations, and two arrows leaving each state. In a machine with *n* inputs, we would have 2^n arrows leaving each state. This is messy if *n* is large. Later, in Figure 7-44, we'll describe a convention whereby a state needn't have one arrow leaving it for each input combination, only one arrow for each different next state.

SUGGESTIVE DRAWINGS

Using the transition, state, and output tables, we can construct a timing diagram that shows the behavior of a state machine for any desired starting state and input sequence. For example, Figure 7-42 shows the behavior of our example machine with a starting state of 00 (A) and a particular pattern on the EN input.

Notice that the value of the EN input affects the next state only at the rising edge of the CLOCK input; that is, the counter counts only if EN = 1 at the rising edge of CLOCK. On the other hand, since MAX is a Mealy-type output, its value is affected by EN at all times. If we also provide a Moore-type output MAXS as suggested in the text, its value depends only on state as shown in the figure.

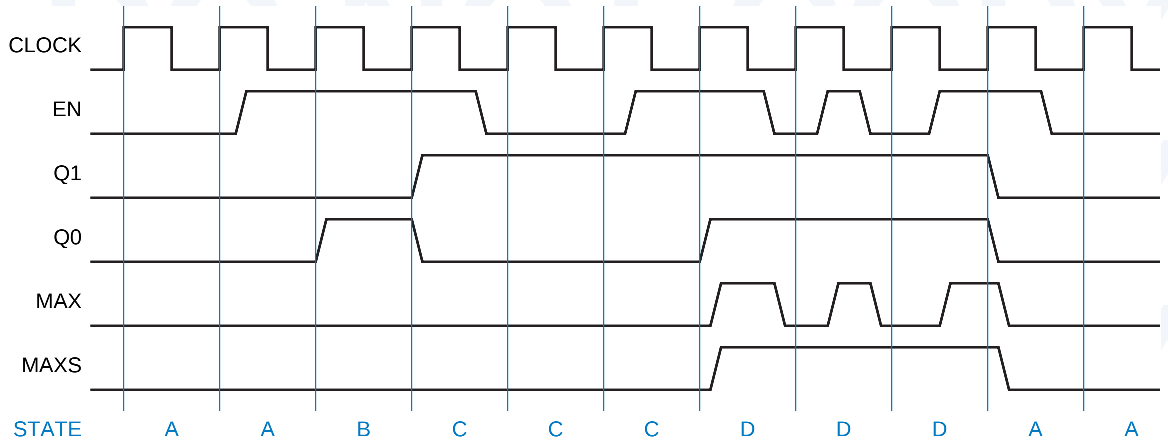
The timing diagram is drawn in a way that shows changes in the MAX and MAXS outputs occurring slightly later than the state and input changes that cause them, reflecting the combinational-logic delay of the output circuits. Naturally, the drawings are merely suggestive; precise timing is normally indicated by a timing table of the type suggested in Section 5.2.1.

the new diagram is that we have used the flip-flops' QN outputs (which are normally the complement of Q) to save a couple of inverters.

In summary, the detailed steps for analyzing a clocked synchronous state machine are as follows:

1. Determine the excitation equations for the flip-flop control inputs. *excitation equations*
2. Substitute the excitation equations into the flip-flop characteristic equations to obtain transition equations. *transition equations*
3. Use the transition equations to construct a transition table. *transition table*
4. Determine the output equations. *output equations*

Figure 7-42 Timing diagram for example state machine.



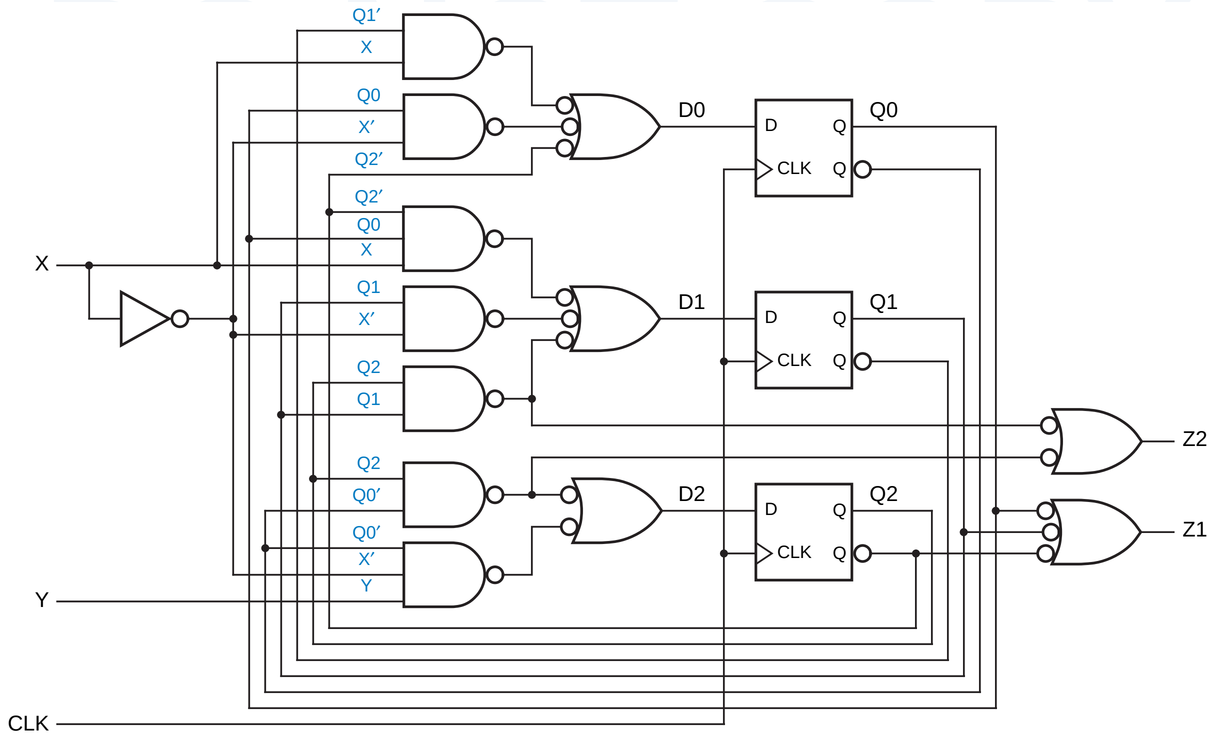


Figure 7-43 A clocked synchronous state machine with three flip-flops and eight states.

5. Add output values to the transition table for each state (Moore) or state/input combination (Mealy) to create a transition/output table. *transition/output table*
6. Name the states and substitute state names for state-variable combinations in the transition/output table to obtain a state/output table. *state names* *state/output table*
7. (Optional) Draw a state diagram corresponding to the state/output table. *state diagram*

We'll go through this complete sequence of steps to analyze another clocked synchronous state machine, shown in Figure 7-43. Reading the logic diagram, we find that the excitation equations are as follows:

$$\begin{aligned} D0 &= Q1' \cdot X + Q0 \cdot X' + Q2 \\ D1 &= Q2' \cdot Q0 \cdot X + Q1 \cdot X' + Q2 \cdot Q1 \\ D2 &= Q2 \cdot Q0' + Q0' \cdot X' \cdot Y \end{aligned}$$

Substituting into the characteristic equation for D flip-flops, we obtain the transition equations:

$$\begin{aligned} Q0* &= Q1' \cdot X + Q0 \cdot X' + Q2 \\ Q1* &= Q2' \cdot Q0 \cdot X + Q1 \cdot X' + Q2 \cdot Q1 \\ Q2* &= Q2 \cdot Q0' + Q0' \cdot X' \cdot Y \end{aligned}$$

Table 7-4 Transition/output and state/output tables for the state machine in Figure 7-43.

(a)

	X Y				
Q2 Q1 Q0	*00*	*01*	*10*	*11*	*Z1 Z2*
000	000	100	001	001	10
001	001	001	011	011	10
010	010	110	000	000	10
011	011	011	010	010	00
100	101	101	101	101	11
101	001	001	001	001	10
110	111	111	111	111	11
111	011	011	011	011	11
	Q2∗ Q1∗ Q0∗				

(b)

	X Y				
S	*00*	*01*	*10*	*11*	*Z1 Z2*
A	A	E	B	B	10
B	B	B	D	D	10
C	C	G	A	A	10
D	D	D	C	C	00
E	F	F	F	F	11
F	B	B	B	B	10
G	H	H	H	H	11
H	D	D	D	D	11
	S∗				

A transition table based on these equations is shown in Table 7-4(a). Reading the logic diagram, we can write two output equations:

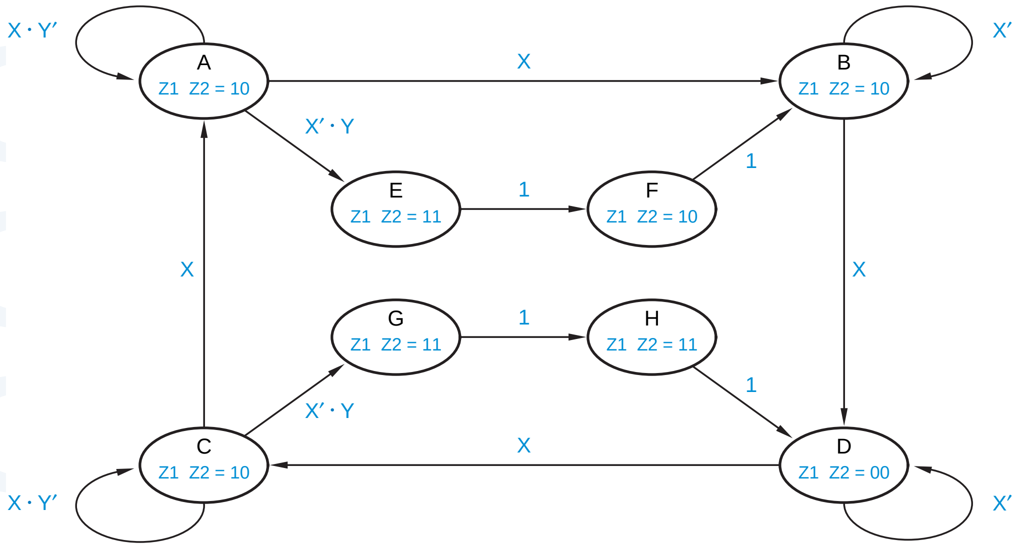
$$Z1 = Q2 + Q1' + Q0'$$
$$Z2 = Q2 \cdot Q1 + Q2 \cdot Q0'$$

The resulting output values are shown in the last column of (a). Assigning state names A–H, we obtain the state/output table shown in (b).

transition expression

A state diagram for the example machine is shown in Figure 7-44. Since our example is a Moore machine, the output values are written with each state. Each arc is labeled with a *transition expression*; a transition is taken for input combinations for which the transition expression is 1. Transitions labeled "1" are always taken.

Figure 7-44 State diagram corresponding to Table 7-4.



The transition expressions on arcs leaving a particular state must be mutually exclusive and all inclusive, as explained below:

- No two transition expressions can equal 1 for the same input combination, since a machine can't have two next states for one input combination. *mutual exclusion*
- For every possible input combination, some transition expression must equal 1, so that all next states are defined. *all inclusion*

Starting with the state table, a transition expression for a particular current state and next state can be written as a sum of minterms for the input combinations that cause that transition. If desired, the expression can then be minimized to give the information in a more compact form. Transition expressions are most useful in the *design* of state machines, where the expressions may be developed from the word description of the problem, as we'll show in \secref{diagdsgn}.

*7.3.5 Analysis of State Machines with J-K Flip-Flops

Clocked synchronous state machines built from J-K flip-flops can also be analyzed by the basic procedure in the preceding subsection. The only difference is that there are two excitation equations for each flip-flop—one for J and the other for K. To obtain the transition equations, both of these must be substituted into the J-K's characteristic equation, $Q* = J \cdot Q' + K' \cdot Q$.

Figure 7-45 is an example state machine using J-K flip-flops. Reading the logic diagram, we can derive the following excitation equations:

$$\begin{aligned} J0 &= X \cdot Y' \\ K0 &= X \cdot Y' + Y \cdot Q1 \\ J1 &= X \cdot Q0 + Y \\ K1 &= Y \cdot Q0' + X \cdot Y' \cdot Q0 \end{aligned}$$

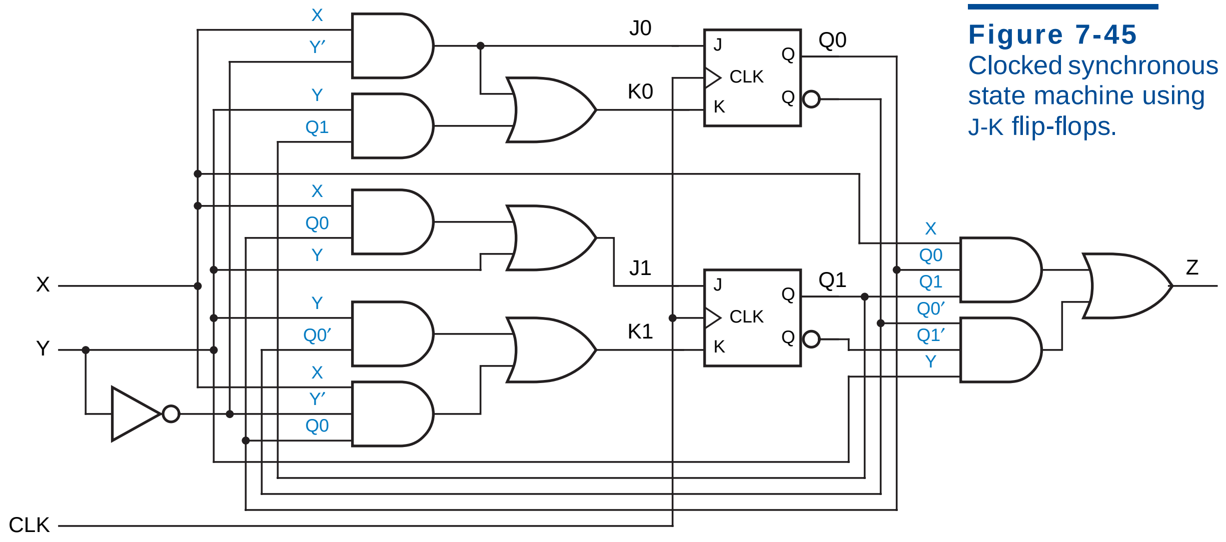


Figure 7-45 Clocked synchronous state machine using J-K flip-flops.

Table 7-5 Transition/output and state/output tables for the state machine in Figure 7-45.

(a)

Q1 Q0	XY 00	01	10	11
00	00, 0	10, 1	01, 0	10, 1
01	01, 0	11, 0	10, 0	11, 0
10	10, 0	00, 0	11, 0	00, 0
11	11, 0	10, 0	00, 1	10, 1
	Q1∗ Q0∗, Z			

(b)

S	XY 00	01	10	11
A	A, 0	C, 1	B, 0	C, 1
B	B, 0	D, 0	C, 0	D, 0
C	C, 0	A, 0	D, 0	A, 0
D	D, 0	C, 0	A, 1	C, 1
	S∗, Z			

Substituting into the characteristic equation for J-K flip-flops, we obtain the transition equations:

$$
\begin{aligned}
Q0* &= J0 \cdot Q0' + K0' \cdot Q0 \\
&= X \cdot Y' \cdot Q0' + (X \cdot Y' + Y \cdot Q1)' \cdot Q0 \\
&= X \cdot Y' \cdot Q0' + X' \cdot Y' \cdot Q0 + X' \cdot Q1' \cdot Q0 + Y \cdot Q1' \cdot Q0 \\
Q1* &= J1 \cdot Q1' + K1' \cdot Q1 \\
&= (X \cdot Q0 + Y) \cdot Q1' + (Y \cdot Q0' + X \cdot Y' \cdot Q0)' \cdot Q1 \\
&= X \cdot Q1' \cdot Q0 + Y \cdot Q1' + X' \cdot Y' \cdot Q1 + Y' \cdot Q1 \cdot Q0' + X' \cdot Q1 \cdot Q0 + Y \cdot Q1 \cdot Q0
\end{aligned}
$$

A transition table based on these equations is shown in Table 7-5(a). Reading the logic diagram, we can write the output equation:

$$Z = X \cdot Q1 \cdot Q0 + Y \cdot Q1' \cdot Q0'$$

The resulting output values are shown in each column of (a) along with the next state. Assigning state names A–D, we obtain the state/output table shown in (b). A corresponding state diagram that uses transition expressions is shown in Figure 7-46.

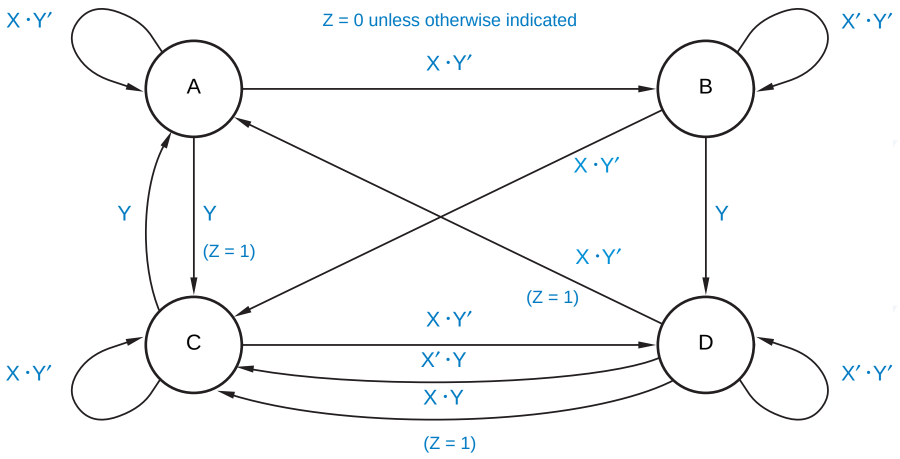


Figure 7-46 State diagram corresponding to the state machine of Table 7-5.

7.4 Clocked Synchronous State-Machine Design

The steps for designing a clocked synchronous state machine, starting from a word description or specification, are just about the reverse of the analysis steps that we used in the preceding section:

1. Construct a state/output table corresponding to the word description or specification, using mnemonic names for the states. (It's also possible to start with a state diagram; this method is discussed in \secref{diagdsgn}.) *state/output table*
2. (Optional) Minimize the number of states in the state/output table. *state minimization*
3. Choose a set of state variables and assign state-variable combinations to the named states. *state assignment*
4. Substitute the state-variable combinations into the state/output table to create a transition/output table that shows the desired next state-variable combination and output for each state/input combination. *transition/output table*
5. Choose a flip-flop type (e.g., D or J-K) for the state memory. In most cases, you'll already have a choice in mind at the outset of the design, but this step is your last chance to change your mind.
6. Construct an excitation table that shows the excitation values required to obtain the desired next state for each state/input combination. *excitation table*
7. Derive excitation equations from the excitation table. *excitation equations*
8. Derive output equations from the transition/output table. *output equations*
9. Draw a logic diagram that shows the state-variable storage elements and realizes the required excitation and output equations. (Or realize the equations directly in a programmable logic device.) *logic diagram*

In this section, we'll describe each of these basic steps in state-machine design. Step 1 is the most important, since it is here that the designer really *designs*, going through the creative process of translating a (perhaps ambiguous) English-language description of the state machine into a formal tabular description. Step 2 is hardly ever performed by experienced digital designers, but designers bring much of their experience to bear in step 3. *design*

Once the first three steps are completed, all of the remaining steps can be completed by "turning the crank," that is, by following a well-defined synthesis procedure. Steps 4 and 6–9 are the most tedious, but they are easily automated. For example, when you design a state machine that will be realized in a programmable logic device, you can use an ABEL compiler to do the cranking, as shown in Section 7.11.2. Still, it's important for you to understand the details of the synthesis procedure, both to give you an appreciation of the compiler's function and to give you a chance of figuring out what's really going on when the compiler produces unexpected results. Therefore, all nine steps of the state-machine design procedure are discussed in the remainder of this section.

STATE-TABLE DESIGN AS A KIND OF PROGRAMMING

Designing a state table (or equivalently, a state diagram) is a creative process that is like writing a computer program in many ways:

- You start with a fairly precise description of inputs and outputs, but a possibly ambiguous description of the desired relationship between them, and usually no clue about how to actually obtain the desired outputs from the inputs.
- During the design, you may have to identify and choose among different ways of doing things, sometimes using common sense, and sometimes arbitrarily.
- You may have to identify and handle special cases that weren't included in the original description.
- You will probably have to keep track of several ideas in your head during the design process.
- Since the design process is not an algorithm, there's no guarantee that you can complete the state table or program using a finite number of states or lines of code. However, unless you work for the government, you must try to do so.
- When you finally run the state machine or program, it will do exactly what you told it to do—no more, no less.
- There's no guarantee that the thing will work the first time; you may have to debug and iterate on the whole process.

Although state-table design is a challenge, there's no need to be intimidated. If you've made it this far in your education, then you've probably written a few programs that worked, and you can become just as good at designing state tables.

7.4.1 State-Table Design Example

There are several different ways to describe a state machine's state table. Later, we'll see how ABEL and VHDL can specify state tables indirectly. In this section, however, we deal only with state tables that are specified directly, in the same tabular format that we used in the previous section for analysis.

We'll present the state-table design process, as well as the synthesis procedure in later subsections, using the simple design problem below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> – A had the same value at each of the two previous clock ticks, *or*
>
> – B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

If the meaning of this specification isn't crystal clear to you at this point, don't worry. Part of your job as a designer is to convert such a specification into a state table that is absolutely unambiguous; even if it doesn't match what was originally intended, it at least forms a basis for further discussion and refinement.

REALIZING RELIABLE RESET

For proper system operation, the hardware design of a state machine should ensure that it enters a known initial state on power-up, such as the INIT state in our design example. Most systems have a RESET signal that is asserted during power-up.

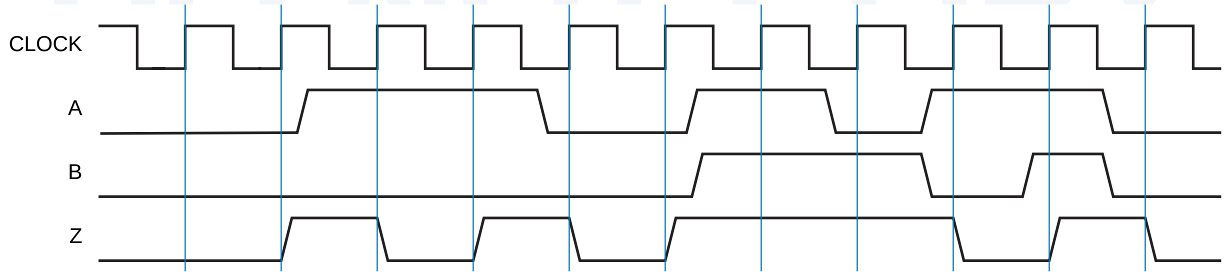
The RESET signal is typically generated by an analog circuit. Such a reset circuit typically detects a voltage (say, 4.5 V) close to the power supply's full voltage, and follows that with a delay (say, 100 ms) to ensure that all components (including oscillators) have had time to stabilize before it "unresets" the system. The Texas Instruments TL7705 is such an analog reset IC; it has an internal 4.5-V reference for the detector and uses an external resistor and capacitor to determine the "unreset" time constant.

If a state machine is built using discrete flip-flops with asynchronous preset and clear inputs, the RESET signal can be applied to these inputs to force the machine into the desired initial state. If preset and clear inputs are not available, or if reset must be synchronous (as in systems using high-speed microprocessors), then the RESET signal may be used as another input to the state machine, with all of the next-state entries going to the desired initial state when RESET is asserted.

As an additional "hint" or requirement, state-table design problems often include timing diagrams that show the state machine's expected behavior for one or more sequences of inputs. Such a timing diagram is unlikely to specify unambiguously the machine's behavior for all possible sequences of inputs but, again, it's a good starting point for discussion and a benchmark against which proposed designs can be checked. Figure 7-47 is such a timing diagram for our example state-table design problem.

The first step in the state-table design is to construct a template. From the word description, we know that our example is a Moore machine—its output depends only on the current state, that is, what happened in previous clock periods. Thus, as shown in Figure 7-48(a), we provide one next-state column for each possible input combination and a single column for the output values. The order in which the input combinations are written doesn't affect this part of the

Figure 7-47 Timing diagram for example state machine.



(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT					0
	. . .					
	. . .					
	. . .					
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0					0
Got a 1 on A	A1					0
		S*				

(c)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1					0
Got two equal A inputs	OK					1
		S*				

(d)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK					1
		S*				

Figure 7-48 Evolution of a state table.

process, but we've written them in Karnaugh-map order to simplify the derivation of excitation equations later. In a Mealy machine we would omit the output column and write the output values along with the next-state values under each input combination. The leftmost column is simply an English-language reminder of the meaning of each state or the "history" associated with it.

initial state

The word description isn't specific about what happens when this machine is first started, so we'll just have to improvise. We'll assume that when power is first applied to the system, the machine enters an *initial state*, called INIT in this example. We write the name of the initial state (INIT) in the first row, and leave room for enough rows (states) to complete the design. We can also fill in the value of Z for the INIT state; common sense says it should be 0 because there were *no* inputs beforehand.

Next, we must fill in the next-state entries for the INIT row. The Z output can't be 1 until we've seen at least two inputs on A, so we'll provide two states, A0 and A1, that "remember" the value of A on the previous clock tick, as shown in Figure 7-48(b). In both of these states, Z is 0, since we haven't satisfied the conditions for a 1 output yet. The precise meaning of state A0 is "Got A = 0 on the previous tick, A ≠ 0 on the tick before that, and B ≠ 1 at some time since the previous pair of equal A inputs." State A1 is defined similarly.

At this point we know that our state machine has at least three states, and we have created two more blank rows to fill in. Hmmmm, this isn't such a good trend! In order to fill in the next-state entries for *one* state (INIT), we had to create *two* new states A0 and A1. If we kept going this way, we could end up with 65,535 states by bedtime! Instead, we should be on the lookout for existing states

(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK	?	OK	OK	?	1
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0					1
Two equal, A=1 last	OK1					1
		S*				

(c)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1					1
		S*				

(d)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1	A0	OK0	OK1	OK1	1
		S*				

Figure 7-49 Continued evolution of a state table.

that have the same meaning as new ones that we might otherwise create. Let's see how it goes.

In state A0, we know that input A was 0 at the previous clock tick. Therefore, if A is 0 again, we go to a new state OK with Z = 1, as shown in Figure 7-48(c). If A is 1, then we don't have two equal inputs in a row, so we go to state A1 to remember that we just got a 1. Likewise in state A1, shown in (d), we go to OK if we get a second 1 input in a row, or to A0 if we get a 0.

Once we get into the OK state, the machine description tells us we can stay there as long as B = 1, irrespective of the A input, as shown in Figure 7-49(a). If B = 0, we have to look for two 1s or two 0s in a row on A again. However, we've got a little problem in this case. The current A input may or may not be the second equal input in a row, so we may still be "OK" or we may have to go back to A0 or A1. We defined the OK state too broadly—it doesn't "remember" enough to tell us which way to go.

The problem is solved in Figure 7-49(b) by splitting OK into two states, OK0 and OK1, that "remember" the previous A input. All of the next states for OK0 and OK1 can be selected from existing states, as shown in (c) and (d). For example, if we get A = 0 in OK0, we can just stay in OK0; we don't have to create a new state that "remembers" three 0s in a row, because the machine's description doesn't require us to distinguish that case. Thus, we have achieved "closure" of the state table, which now describes a *finite*-state machine. As a sanity check, Figure 7-50 repeats the timing diagram of Figure 7-47, listing the states that should be visited according to our final state table.

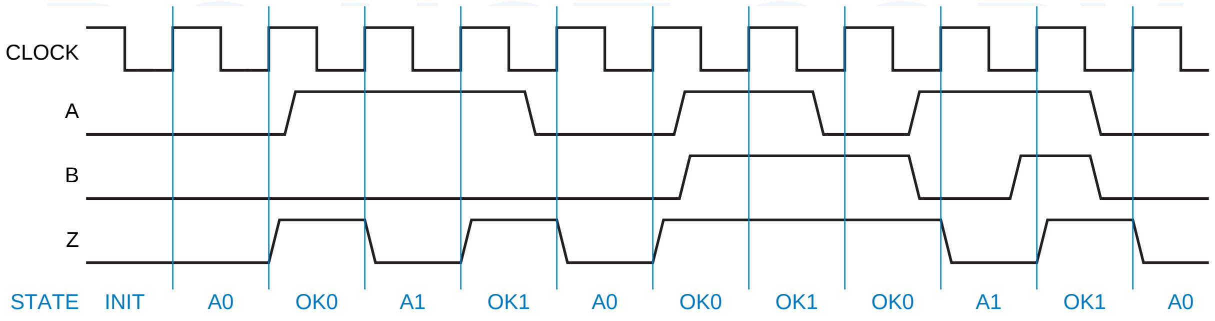


Figure 7-50 Timing diagram and state sequence for example state machine.

7.4.2 State Minimization

Figure 7-49(d) is a "minimal" state table for our original word description, in the sense that it contains the fewest possible states. However, Figure 7-51 shows other state tables, with more states, that also do the job. Formal procedures can be used to minimize the number of states in such tables.

equivalent states

The basic idea of formal minimization procedures is to identify *equivalent states*, where two states are equivalent if it is impossible to distinguish the states by observing only the current and future *outputs* of the machine (and *not* the internal state variables). A pair of equivalent states can be replaced by a single state.

Two states S1 and S2 are equivalent if two conditions are true. First, S1 and S2 must produce the same values at the state-machine output(s); in a Mealy machine, this must be true for all input combinations. Second, for each input combination, S1 and S2 must have either the same next state or equivalent next states.

Thus, a formal state-minimization procedure shows that states OK00 and OKA0 in Figure 7-51(a) are equivalent because they produce the same output

Figure 7-51 Nonminimal state tables equivalent to Figure 7-49(d).

(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK00	OK00	A1	A1	0
Got a 1 on A	A1	A0	A0	OK11	OK11	0
Got 00 on A	OK00	OK00	OK00	OKA1	A1	1
Got 11 on A	OK11	A0	OKA0	OK11	OK11	1
OK, got a 0 on A	OKA0	OK00	OK00	OKA1	A1	1
OK, got a 1 on A	OKA1	A0	OKA0	OK11	OK11	1
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK00	OK00	A1	A1	0
Got a 1 on A	A1	A0	A0	OK11	OK11	0
Got 00 on A	OK00	OK00	OK00	A001	A1	1
Got 11 on A	OK11	A0	A110	OK11	OK11	1
Got 001 on A, B=1	A001	A0	AE10	OK11	OK11	1
Got 110 on A, B=1	A110	OK00	OK00	AE01	A1	1
Got bb...10 on A, B=1	AE10	OK00	OK00	AE01	A1	1
Got bb...01 on A, B=1	AE01	A0	AE10	OK11	OK11	1
		S*				

and their next-state entries are identical. Since the states are equivalent, state OK00 may be eliminated and its occurrences in the table replaced by OKA0, or vice versa. Likewise, states OK11 and OKA1 are equivalent.

To minimize the state table in Figure 7-51(b), a formal procedure must use a bit of circular reasoning. States OK00, A110, and AE10 all produce the same output and have almost identical next-state entries, so they might be equivalent. They are equivalent only if A001 and AE01 are equivalent. Similarly, OK11, A001, and AE01 are equivalent only if A110 and AE10 are equivalent. In other words, the states in the first set are equivalent if the states in the second set are, and vice versa. So, let's just go ahead and say they're equivalent.

IS THIS REALLY ALL NECESSARY?

Details of formal state-minimization procedures are discussed in advanced textbooks, cited in the References. However, these procedures are seldom used by most digital designers. By carefully matching state meanings to the requirements of the problem, experienced digital designers produce state tables for small problems with a minimal or near-minimal number of states, without using a formal minimization procedure. Also, there are situations where *increasing* the number of states may simplify the design or reduce its cost, so even an automated state-minimization procedure doesn't necessarily help. A designer can do more to simplify a state machine during the state-assignment phase of the design, discussed in the next subsection.

7.4.3 State Assignment

The next step in the design process is to determine how many binary variables are required to represent the states in the state table, and to assign a specific combination to each named state. We'll call the binary combination assigned to a particular state a *coded state*. The *total number of states* in a machine with n flip-flops is 2^n, so the number of flip-flops needed to code s states is $\lceil \log_2 s \rceil$, the smallest integer greater than or equal to $\log_2 s$.

coded state

total number of states

For reference, the state/output table of our example machine is repeated in Table 7-6. It has five states, so it requires three flip-flops. Of course, three flip-flops provide a total of eight states, so there will be $8 - 5 = 3$ *unused states*. We'll discuss alternatives for handling the unused states at the end of this subsection. Right now, we have to deal with lots of choices for the five coded states.

unused states

INITIAL VERSUS IDLE STATES

The example state machine in this subsection visits its initial state only during reset. Many machines are designed instead with an "idle" state that is entered both at reset and whenever the machine has nothing in particular to do.

Table 7-6
State and output table for example problem.

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S∗				

The simplest assignment of *s* coded states to 2^n possible states is to use the first *s* binary integers in binary counting order, as shown in the first assignment column of Table 7-7. However, the simplest state assignment does not always lead to the simplest excitation equations, output equations, and resulting logic circuit. In fact, the state assignment often has a major effect on circuit cost, and may interact with other factors, such as the choice of storage elements (e.g., D vs. J-K flip-flops) and the realization approach for excitation and output logic (e.g., sum-of-products, product-of-sums, or ad hoc).

So, how do we choose the best state assignment for a given problem? In general, the only formal way to find the *best* assignment is to try *all* the assignments. That's too much work, even for students. Instead, most digital designers rely on experience and several practical guidelines for making reasonable state assignments:

- Choose an initial coded state into which the machine can easily be forced at reset (00. . . 00 or 11. . . 11 in typical circuits).
- Minimize the number of state variables that change on each transition.

CAUTION: MATH

The number of different ways to choose *m* coded states out of a set of *n* possible states is given by a *binomial coefficient*, denoted $\binom{n}{m}$, whose value is $\frac{n!}{m! \cdot (n-m)}$. (We used binomial coefficients previously in Section 2.10, in the context of decimal coding.) In our example, there are $\binom{8}{5}$ different ways to choose five coded states out of eight possible states, and 5! ways to assign the five named states to each different choice. So there are $\frac{8!}{5! \cdot 3!} \cdot 5!$ or 6,720 different ways to assign the five states of our example machine to combinations of three binary state variables. We don't have time to look at all of them.

State name	Simplest Q1–Q3	Decomposed Q1–Q3	One-hot Q1–Q5	Almost one-hot Q1–Q4
	Assignment			
INIT	000	000	00001	0000
A0	001	100	00010	0001
A1	010	101	00100	0010
OK0	011	110	01000	0100
OK1	100	111	10000	1000

Table 7-7 Possible state assignments for the state machine in Table 7-6.

- Maximize the number of state variables that don't change in a group of related states (i.e., a group of states in which most of transitions stay in the group).
- Exploit symmetries in the problem specification and the corresponding symmetries in the state table. That is, suppose that one state or group of states means almost the same thing as another. Once an assignment has been established for the first, a similar assignment, differing only in one bit, should be used for the second.
- If there are unused states (i.e., if $s < 2^n$ where $n = \lceil \log_2 s \rceil$), then choose the "best" of the available state-variable combinations to achieve the foregoing goals. That is, don't limit the choice of coded states to the first *s* *n*-bit integers.
- Decompose the set of state variables into individual bits or fields where each bit or field has a well-defined meaning with respect to the input effects or output behavior of the machine.
- Consider using more than the minimum number of state variables to make a decomposed assignment possible.

Some of these ideas are incorporated in the "decomposed" state assignment in Table 7-7. As before, the initial state is 000, which is easy to force either asynchronously (applying the RESET signal to the flip-flop CLR inputs) or synchronously (by AND'ing RESET′ with all of the D flip-flop inputs). After this point, the assignment takes advantage of the fact that there are only four states in addition to INIT, which is a fairly "special" state that is never re-entered once the machine gets going. Therefore, Q1 can be used to indicate whether or not the machine is in the INIT state, and Q2 and Q3 can be used to distinguish among the four non-INIT states.

The non-INIT states in the "decomposed" column of Table 7-7 appear to have been assigned in binary counting order, but that's just a coincidence. State bits Q2 and Q3 actually have individual meanings in the context of the state machine's inputs and output. Q3 gives the previous value of A, and Q2 indicates that the conditions for a 1 output are satisfied in the current state. By decomposing the state-bit meanings in this way, we can expect the next-state and output logic to be simpler than in a "random" assignment of Q2,Q3 combinations to the non-INIT states. We'll continue the state-machine design based on this assignment in later subsections.

one-hot assignment

Another useful state assignment, one that can be adapted to any state machine, is the *one-hot assignment* shown in Table 7-7. This assignment uses more than the minimum number of state variables—it uses one bit per state. In addition to being simple, a one-hot assignment has the advantage of usually leading to small excitation equations, since each flip-flop must be set to 1 for transitions into only one state. An obvious disadvantage of a one-hot assignment, especially for machines with many states, is that it requires (many) more than the minimum number of flip-flops. However, the one-hot encoding is ideal for a machine with s states that is required to have a set of 1-out-of-s coded outputs indicating its current state. The one-hot-coded flip-flop outputs can be used directly for this purpose, with no additional combinational output logic.

The last column of Table 7-7 is an "almost one-hot assignment" that uses the "no-hot" combination for the initial state. This makes a lot of sense for two reasons: It's easy to initialize most storage devices to the all-0s state, and the initial state in this machine is never revisited once the machine gets going. Completing the state-machine design using this state assignment is considered in Exercises 7.35 and 7.38.

unused states

We promised earlier to consider the disposition of *unused states* when the number of states available with n flip-flops, 2^n, is greater than the number of states required, s. There are two approaches that make sense, depending on the application requirements:

- *Minimal risk.* This approach assumes that it is possible for the state machine somehow to get into one of the unused (or "illegal") states, perhaps because of a hardware failure, an unexpected input, or a design error. Therefore, all of the unused state-variable combinations are identified, and explicit next-state entries are made so that, for any input combination, the unused states go to the "initial" state, the "idle" state, or some other "safe" state. This is an automatic consequence of some design methodologies if the initial state is coded 00. . . 00.
- *Minimal cost.* This approach assumes that the machine will never enter an unused state. Therefore, in the transition and excitation tables, the next-state entries of the unused states can be marked as "don't-cares." In most cases, this simplifies the excitation logic. However, the machine's behavior if it ever does enter an unused state may be pretty weird.

We'll look at both of these approaches as we complete the design of our example state machine.

7.4.4 Synthesis Using D Flip-Flops

Once we've assigned coded states to the named states of a machine, the rest of the design process is pretty much "turning the crank." In fact, in Section 7.11.2 we'll describe software tools that can turn the crank for you. Just so that you'll appreciate those tools, however, we'll go through the process by hand in this subsection.

Coded states are substituted for named states in the (possibly minimized) state table to obtain a *transition table*. The transition table shows the next coded state for each combination of current coded state and input. Table 7-8 shows the transition and output table that is obtained from the example state machine of Table 7-6 on page 472 using the "decomposed" assignment of Table 7-7 on page 473.

transition table

The next step is to write an *excitation table* that shows, for each combination of coded state and input, the flip-flop excitation input values needed to make the machine go to the desired next coded state. This structure and content of this table depend on the type of flip-flops that are used (D, J-K, T, etc.). We *usually* have a particular flip-flop type in mind at the beginning of a design—and we *certainly* do in this subsection, given its title. In fact, most state-machine designs nowadays use D flip-flops, because of their availability in both discrete packages and programmable logic devices, and because of their ease of use (compare with J-K flip-flops in the next subsection).

excitation table

Of all flip-flop types, a D flip-flop has the simplest characteristic equation, Q∗ = D. Each D flip-flop in a state machine has a single excitation input, D, and the excitation table must show the value required at each flip-flop's D input for each coded-state/input combination. Table 7-9 shows the excitation table for our example problem. Since D = Q∗, the excitation table is identical to the transition table, except for labeling of its entries. Thus, with D flip-flops, you don't really need to write a separate excitation table; you can just call the first table a *transition/excitation table*.

transition/excitation table

The excitation table is like a truth table for three combinational logic functions (D1, D2, D3) of five variables (A, B, Q1, Q2, Q3). Accordingly, we can

Table 7-8 Transition and output table for example problem.

	A B				
Q1 Q2 Q3	***00***	***01***	***11***	***10***	***Z***
000	100	100	101	101	0
100	110	110	101	101	0
101	100	100	111	111	0
110	110	110	111	101	1
111	100	110	111	111	1
	Q1∗ Q2∗ Q3∗				

Table 7-9 Excitation and output table for Table 7-8 using D flip-flops.

	AB				
Q1 Q2 Q3	**00**	**01**	**11**	**10**	**Z**
000	100	100	101	101	0
100	110	110	101	101	0
101	100	100	111	111	0
110	110	110	111	101	1
111	100	110	111	111	1
	D1 D2 D3				

design circuits to realize these functions using any of the combinational design methods at our disposal. In particular, we can transfer the information in the excitation table to Karnaugh maps, which we may call *excitation maps*, and find a minimal sum-of-products or product-of-sums expression for each function.

excitation maps

Excitation maps for our example state machine are shown in Figure 7-52. Each function, such as D1, has five variables and therefore uses a *5-variable Karnaugh map*. A 5-variable map is drawn as a pair of 4-variable maps, where cells in the same position in the two maps are considered to be adjacent. These maps are a bit unwieldy, but if you want to design by hand any but the most trivial state machines, you're going to get stuck with 5-variable maps and worse. At least we had the foresight to label the input combinations of the original state table in Karnaugh-map order, which makes it easier to transfer information to the maps in this step. However, note that the *states* were not assigned in Karnaugh-map order; in particular, the rows for states 110 and 111 are in the opposite order in the map as in the excitation table.

5-variable Karnaugh map

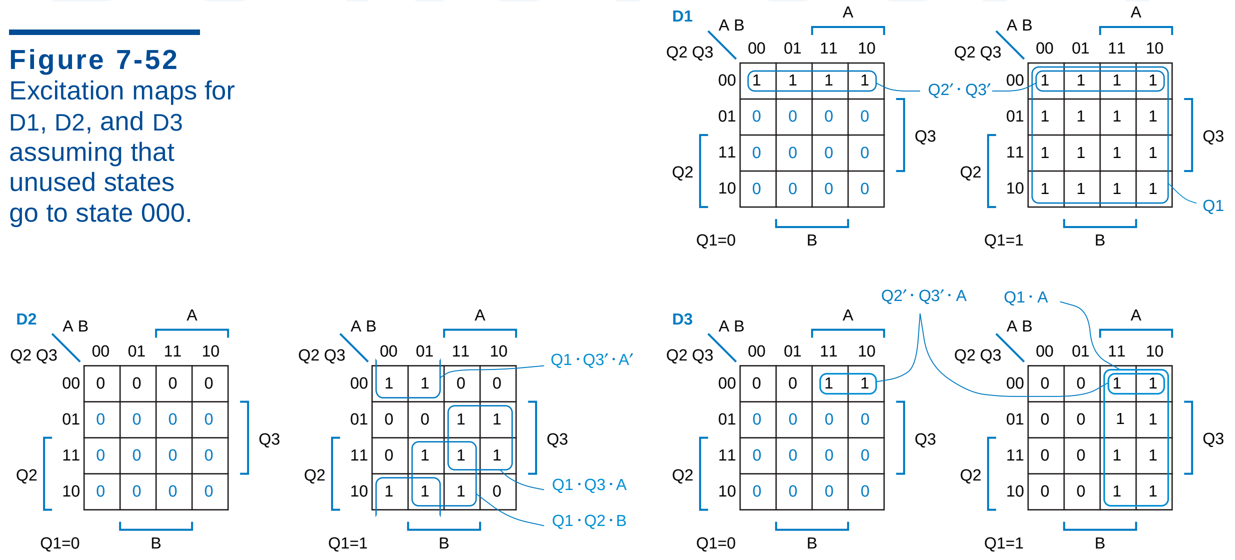


Figure 7-52 Excitation maps for D1, D2, and D3 assuming that unused states go to state 000.

It is in this step, transferring the excitation table to excitation maps, that we discover why the excitation table is not quite a truth table—it does not specify functional values for *all* input combinations. In particular, the next-state information for the unused states, 001, 010, and 011, is not specified. Here we must make a choice, discussed in the preceding subsection, between a minimal-risk and a minimal-cost strategy for handling the unused states. Figure 7-52 has taken the minimal-risk approach: The next state for each unused state and input combination is 000, the INIT state. The three rows of colored 0s in each Karnaugh map are the result of this choice. With the maps completely filled in, we can now obtain minimal sum-of-products expressions for the flip-flop excitation inputs:

$$\begin{aligned} D1 &= Q1 + Q2' \cdot Q3' \\ D2 &= Q1 \cdot Q3' \cdot A' + Q1 \cdot Q3 \cdot A + Q1 \cdot Q2 \cdot B \\ D3 &= Q1 \cdot A + Q2' \cdot Q3' \cdot A \end{aligned}$$

An output equation can easily be developed directly from the information in Table 7-9. The output equation is simpler than the excitation equations, because the output is a function of state only. We could use a Karnaugh map, but it’s easy to find a minimal-risk output function algebraically, by writing it as the sum of the two coded states (110 and 111) in which Z is 1:

$$\begin{aligned} Z &= Q1 \cdot Q2 \cdot Q3' + Q1 \cdot Q2 \cdot Q3 \\ &= Q1 \cdot Q2 \end{aligned}$$

At this point, we’re just about done with the state-machine design. If the state machine is going to be built with discrete flip-flops and gates, then the final step is to draw a logic diagram. On the other hand, if we are using a programmable logic device, then we only have to enter the excitation and output equations into a computer file that specifies how to program the device, as an example shows in Section 7.11.1. Or, if we’re lucky, we specified the machine using a state-machine description language like ABEL in the first place (Section 7.11.2), and the computer did all the work in this subsection for us!

MINIMAL-COST SOLUTION

If we choose in our example to derive minimal-cost excitation equations, we write “don’t-cares” in the next-state entries for the unused states. The colored d’s in Figure 7-53 are the result of this choice. The excitation equations obtained from this map are somewhat simpler than before:

$$\begin{aligned} D1 &= 1 \\ D2 &= Q1 \cdot Q3' \cdot A' + Q3 \cdot A + Q2 \cdot B \\ D3 &= A \end{aligned}$$

For a minimal-cost output function, the value of Z is a “don’t-care” for the unused states. This leads to an even simpler output function, $Z = Q2$. The logic diagram for the minimal-cost solution is shown in Figure 7-54.

Figure 7-53
Excitation maps for D1, D2, and D3 assuming that next states of unused states are "don't-cares."

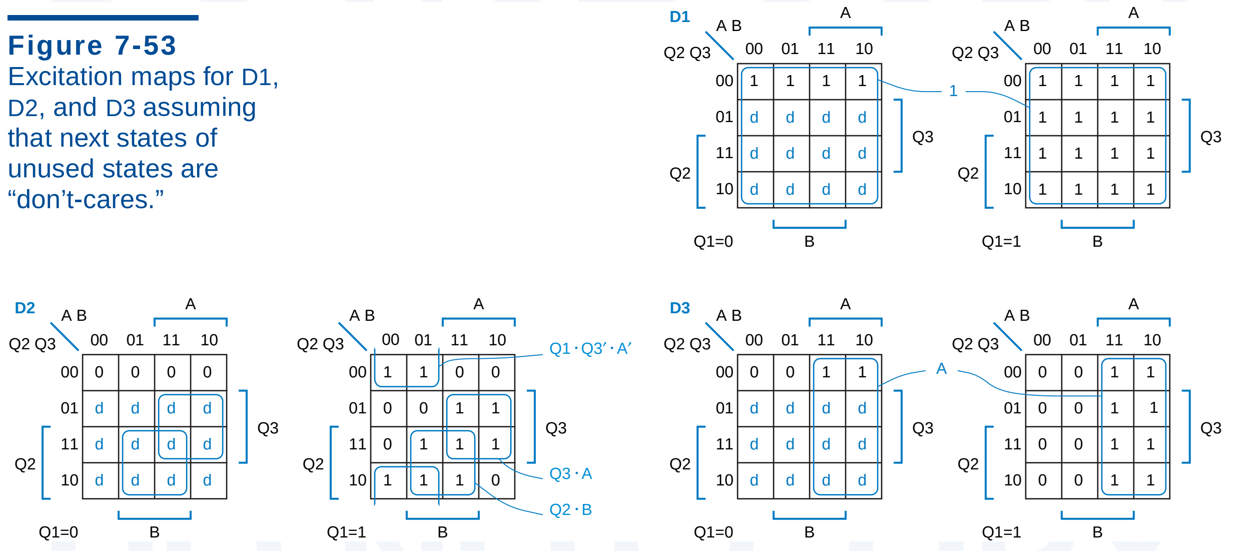
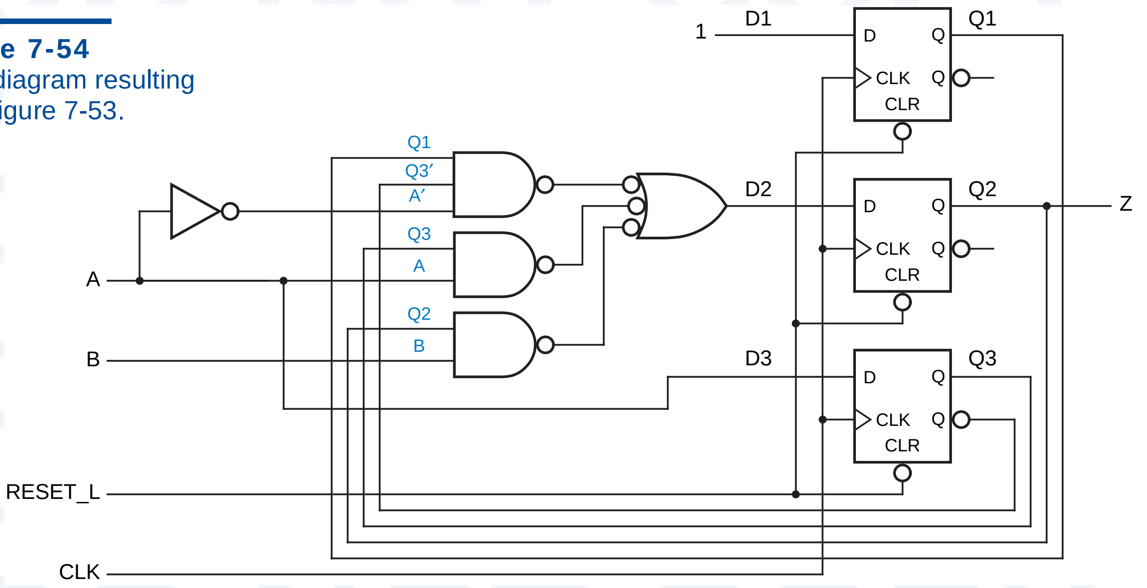


Figure 7-54
Logic diagram resulting from Figure 7-53.



*7.4.5 Synthesis Using J-K Flip-Flops

At one time, J-K flip-flops were popular for discrete SSI state-machine designs, since a J-K flip-flop embeds more functionality than a D flip-flop in the same size SSI package. By "more functionality" we mean that the combination of J and K inputs yields more possibilities for controlling the flip-flop than a single D input does. As a result, a state machine's excitation logic may be simpler using J-K flip-flops than using D flip-flops, which reduced package count when SSI gates were used for the excitation logic.

JUST FOR FUN

While minimizing excitation logic was a big deal in the days of SSI-based design, the name of the game has changed with PLDs and ASICs. As you might guess from your knowledge of the AND-OR structure of combinational PLDs, the need to provide separate AND-OR arrays for the J and K inputs of a J-K flip-flop would be a distinct disadvantage in a sequential PLD.

In ASIC technologies, J-K flip-flops aren't so desirable either. For example, in LSI Logic Corp.'s LCA10000 series of CMOS gate arrays, an FD1 D flip-flop macrocell uses 7 "gate cells", while an FJK1 J-K flip-flop macrocell uses 9 gate cells, over 25% more chip area. Therefore, a more cost-effective design usually results from sticking with D flip-flops and using the extra chip area for more complex excitation logic in just the cases where it's really needed.

Still, this subsection describes the J-K synthesis process "just for fun."

Up through the state-assignment step, the design procedure with J-K flip-flops is basically the same as with D flip-flops. The only difference is that a designer might select a slightly different state assignment, knowing the sort of behavior that can easily be obtained from J-K flip-flops (e.g., "toggling" by setting J and K to 1).

The big difference occurs in the derivation of an excitation table from the transition table. With D flip-flops, the two tables are identical; using the D's characteristic equation, $Q* = D$, we simply substitute $D = Q*$ for each entry. With J-K flip-flops, each entry in the excitation table has twice as many bits as in the transition table, since there are two excitation inputs per flip-flop.

J-K application table

A J-K flip-flop's characteristic equation, $Q* = J \cdot Q' + K' \cdot Q$, cannot be rearranged to obtain independent equations for J and K. Instead, the required values for J and K are expressed as functions of Q and Q* in a J-K *application table*, Table 7-10. According to the first row, if Q is currently 0, all that is required to obtain 0 as the next value of Q is to set J to 0; the value of K doesn't matter. Similarly, according to the third row, if Q is currently 1, the next value of Q will be 0 if K is 1, regardless of J's value. Each desired transition can be obtained by either of two different combinations on the J and K inputs, so we get a "don't-care" entry in each row of the application table.

To obtain a J-K excitation table, the designer must look at both the current and desired next value of each state bit in the transition table and substitute the

Q	*Q**	*J*	*K*
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

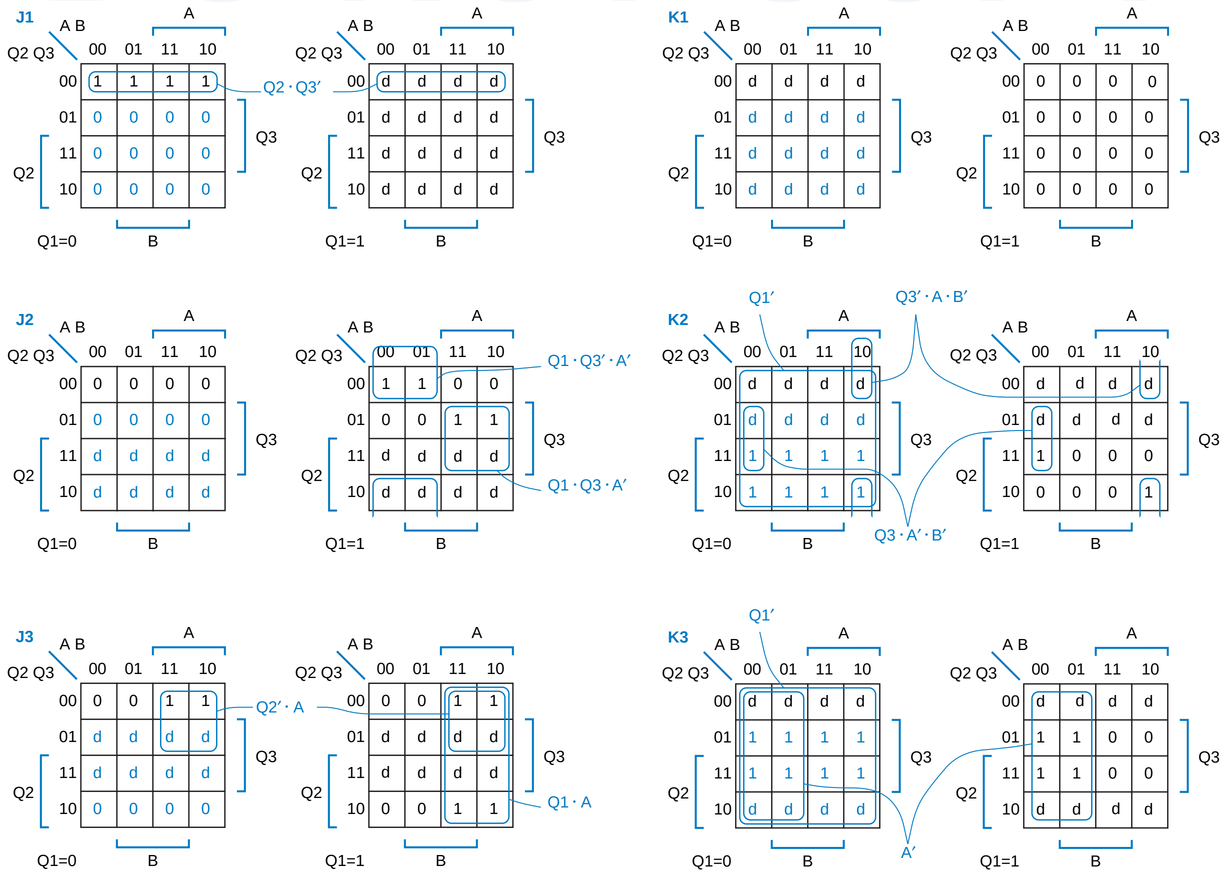
Table 7-10 Application table for J-K flip-flops.

corresponding pair of J and K values from the application table. For the transition table in Table 7-8 on page 475, these substitutions produce the excitation table in Table 7-11. For example, in state 100 under input combination 00, Q1 is 1 and the required Q1∗ is 1; therefore, “d0” is entered for J1 K1. For the same state/input combination, Q2 is 0 and Q2∗ is 1, so “1d” is entered for J2 K2. Obviously, it takes quite a bit of patience and care to fill in the entire excitation table (a job best left to a computer).

As in the D synthesis example of the preceding subsection, the excitation table is *almost* a truth table for the excitation functions. This information is transferred to Karnaugh maps in Figure 7-55.

The excitation table does not specify next states for the unused states, so once again we must choose between the minimal-risk and minimal-cost approaches. The colored entries in the Karnaugh maps result from taking the minimal-risk approach.

Figure 7-55 Excitation maps for J1, K1, J2, K2, J3, and K3, assuming that unused states go to state 000.



	AB				
Q1 Q2 Q3	00	01	11	10	Z
000	1d, 0d, 0d	1d, 0d, 0d	1d, 0d, 1d	1d, 0d, 1d	0
100	d0, 1d, 0d	d0, 1d, 0d	d0, 0d, 1d	d0, 0d, 1d	0
101	d0, 0d, d1	d0, 0d, d1	d0, 1d, d0	d0, 1d, d0	0
110	d0, d0, 0d	d0, d0, 0d	d0, d0, 1d	d0, d1, 1d	1
111	d0, d1, d1	d0, d0, d1	d0, d0, d0	d0, d0, d0	1
	J1 K1, J2 K2, J3 K3				

Table 7-11 Excitation and output table for the state machine of Table 7-8, using J-K flip-flops.

Note that even though the "safe" next state for unused states is 000, we didn't just put 0s in the corresponding map cells, as we were able to do in the D case. Instead, we still had to work with the application table to determine the proper combination of J and K needed to get Q∗ = 0 for each unused state entry, once again a tedious and error-prone process.

Using the maps in Figure 7-55, we can derive sum-of-products excitation equations:

$$\begin{aligned} J1 &= Q2' \cdot Q3' & K1 &= 0 \\ J2 &= Q1 \cdot Q3' \cdot A' + Q1 \cdot Q3 \cdot A & K2 &= Q1' + Q3' \cdot A \cdot B' + Q3 \cdot A' \cdot B' \\ J3 &= Q2' \cdot A + Q1 \cdot A & K3 &= Q1' + A' \end{aligned}$$

These equations take two more gates to realize than do the preceding subsection's minimal-risk equations using D flip-flops, so J-K flip-flops didn't save us anything in this example, least of all design time.

MINIMAL-COST SOLUTION

In the preceding design example, excitation maps for the minimal-cost approach would have been somewhat easier to construct, since we could have just put d's in all of the unused state entries. Sum-of-products excitation equations obtained from the minimal-cost maps (not shown) are as follows:

$$\begin{aligned} J1 &= 1 & K1 &= 0 \\ J2 &= Q1 \cdot Q3' \cdot A' + Q3 \cdot A & K2 &= Q3' \cdot A \cdot B' + Q3 \cdot A' \cdot B' \\ J3 &= A & K3 &= A' \end{aligned}$$

The state encoding for the J-K circuit is the same as in the D circuit, so the output equation is the same, Z = Q1 · Q2 for minimal risk, Z = Q2 for minimal cost.

A logic diagram corresponding to the minimal-cost equations is shown in Figure 7-56. This circuit has two more gates than the minimal-cost D circuit in Figure 7-54, so J-K flip-flops still didn't save us anything.

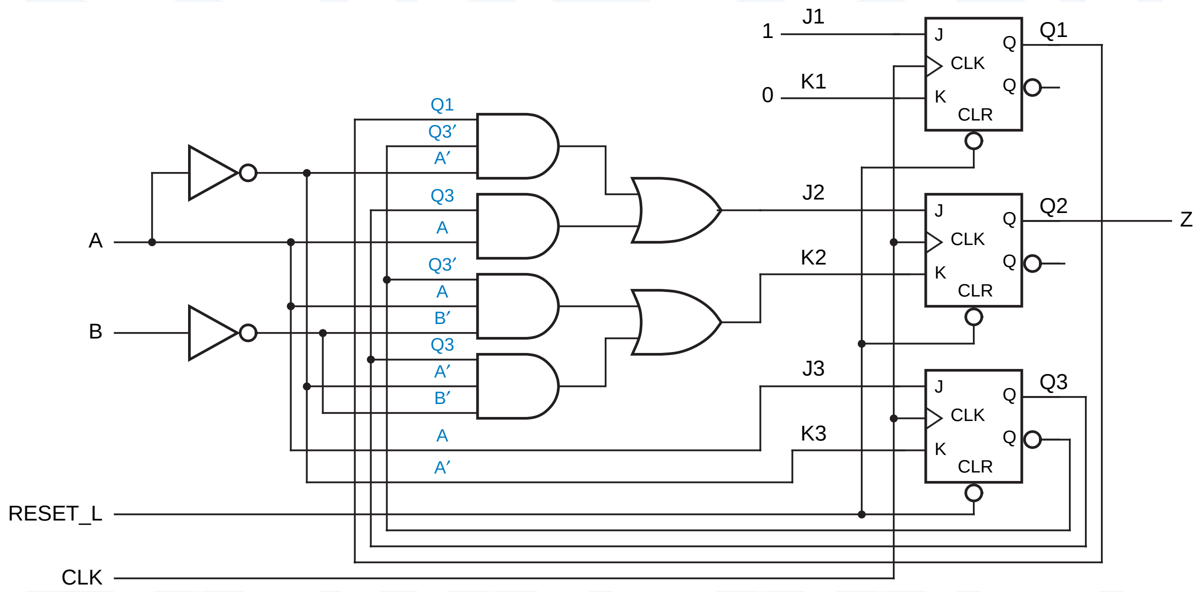


Figure 7-56 Logic diagram for example state machine using J-K flip-flops and minimal-cost excitation logic.

7.4.6 More Design Examples Using D Flip-Flops

We'll conclude this section with two more state-machine design examples using D flip-flops. The first example is a "1s-counting machine":

> Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z. The output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise.

At first glance, you might think the machine needs an infinite number of states, since it counts 1 inputs over an arbitrarily long time. However, since the output indicates the number of inputs received *modulo 4*, four states are sufficient. We'll name them S0–S3, where S0 is the initial state and the total number of 1s received in S*i* is *i* modulo 4. Table 7-12 is the resulting state and output table.

Table 7-12 State and output table for 1s-counting machine.

		X Y				
Meaning	***S***	***00***	***01***	***11***	***10***	***Z***
Got zero 1s (modulo 4)	S0	S0	S1	S2	S1	1
Got one 1 (modulo 4)	S1	S1	S2	S3	S2	0
Got two 1s (modulo 4)	S2	S2	S3	S0	S3	0
Got three 1s (modulo 4)	S3	S3	S0	S1	S0	0
			S*			

	XY				
Q1 Q2	***00***	***01***	***11***	***10***	***Z***
00	00	01	11	01	1
01	01	11	10	11	0
11	11	10	00	10	0
10	10	00	01	00	0
	Q1∗ Q2∗ or D1 D2				

Table 7-13
Transition/excitation and output table for 1s-counting machine.

The 1s-counting machine can use two state variables to code its four states, with no unused states. In this case, there are only 4! possible assignments of coded states to named states. Still, we'll try only one of them. We'll assign coded states to the named states in Karnaugh-map order (00, 01, 11, 10) for two reasons: In this state table, it minimizes the number of state variables that change for most transitions, potentially simplifying the excitation equations; and it simplifies the mechanical transfer of information to excitation maps.

A transition/excitation table based on our chosen state assignment is shown in Table 7-13. Since we're using D flip-flops, the transition and excitation tables are the same. Corresponding Karnaugh maps for D1 and D2 are shown in Figure 7-57. Since there are no unused states, all of the information we need is in the excitation table; no choice is required between minimal-risk and minimal-cost approaches. The excitation equations can be read from the maps, and the output equation can be read directly from the transition/excitation table:

$$\begin{aligned} D1 &= Q2 \cdot X' \cdot Y + Q1' \cdot X \cdot Y + Q1 \cdot X' \cdot Y' + Q2 \cdot X \cdot Y' \\ D2 &= Q1' \cdot X' \cdot Y + Q1' \cdot X \cdot Y' + Q2 \cdot X' \cdot Y' + Q2' \cdot X \cdot Y \\ Z &= Q1' \cdot Q2' \end{aligned}$$

A logic diagram using D flip-flops and AND-OR or NAND-NAND excitation logic can be drawn from these equations.

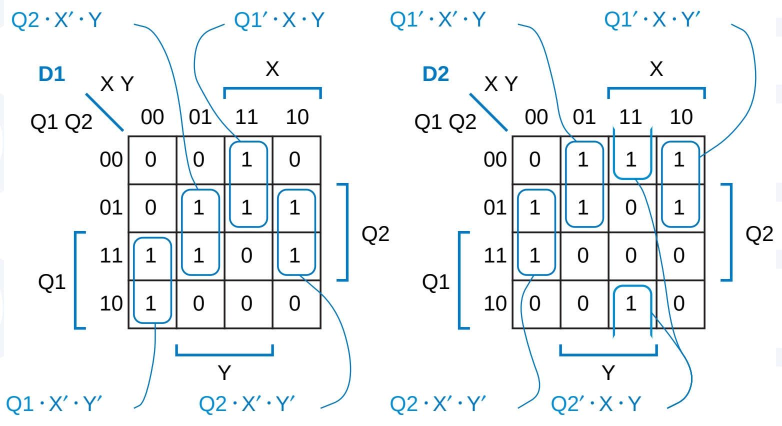


Figure 7-57
Excitation maps for D1 and D2 inputs in 1s-counting machine.

The second example is a "combination lock" state machine that activates an "unlock" output when a certain binary input sequence is received:

> Design a clocked synchronous state machine with one input, X, and two outputs, UNLK and HINT. The UNLK output should be 1 if and only if X is 0 and the sequence of inputs received on X at the preceding seven clock ticks was 0110111. The HINT output should be 1 if and only if the current value of X is the correct one to move the machine closer to being in the "unlocked" state (with UNLK = 1).

It should be apparent from word description that this is a Mealy machine. The UNLK output depends on both the past history of inputs and X's current value, and HINT depends on both the state and the current X (indeed, if the current X produces HINT = 0, then the clued-in user will want to change X before the clock tick).

A state and output table for the combination lock is presented in Table 7-14. In the initial state, A, we assume that we have received no inputs in the required sequence; we're looking for the first 0 in the sequence. Therefore, as long as we get 1 inputs, we stay in state A, and we move to state B when we receive a 0. In state B, we're looking for a 1. If we get it, we move on to C; if we don't, we can stay in B, since the 0 we just received might still turn out to be the first 0 in the required sequence. In each successive state, we move on to the next state if we get the correct input, and we go back to A or B if we get the wrong one. An exception occurs in state G; if we get the wrong input (a 0) there, the previous three inputs might still turn out to be the first three inputs of the required sequence, so we go back to state E instead of B. In state H, we've received the required sequence, so we set UNLK to 1 if X is 0. In each state, we set HINT to 1 for the value of X that moves us closer to state H.

Table 7-14 State and output table for combination-lock machine.

		X	
Meaning	***S***	***0***	***1***
Got zip	A	B, 01	A, 00
Got 0	B	B, 00	C, 01
Got 01	C	B, 00	D, 01
Got 011	D	E, 01	A, 00
Got 0110	E	B, 00	F, 01
Got 01101	F	B, 00	G, 01
Got 011011	G	E, 00	H, 01
Got 0110111	H	B, 11	A, 00
		S*, UNLK HINT	

Table 7-15 Transition/excitation table for combination-lock machine.

Q1 Q2 Q3	X = 0	X = 1
000	001, 01	000, 00
001	001, 00	010, 01
010	001, 00	011, 01
011	100, 01	000, 00
100	001, 00	101, 01
101	001, 00	110, 01
110	100, 00	111, 01
111	001, 11	000, 00
	Q1∗ Q2∗ Q3∗, UNLK HINT	

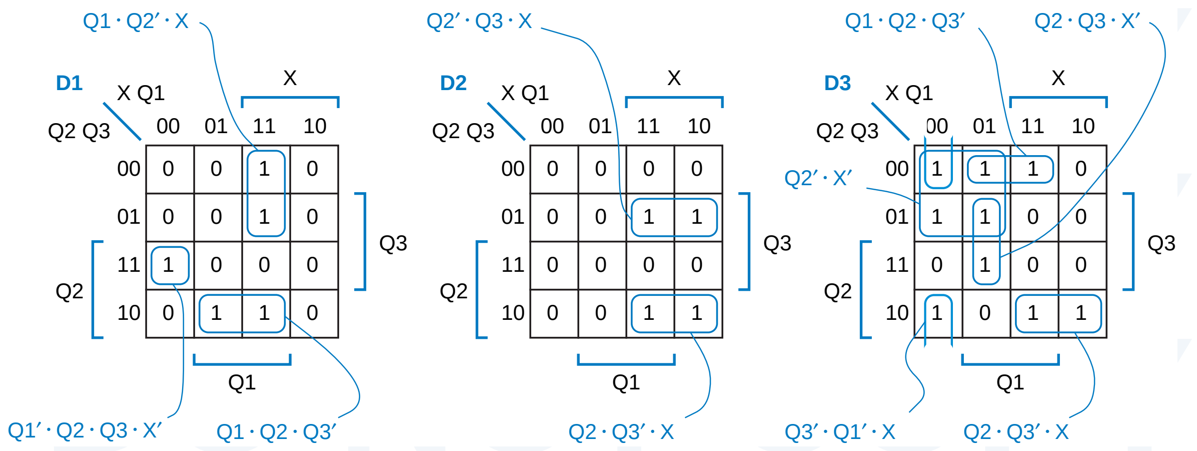
The combination lock's eight states can be coded with three state variables, leaving no unused states. There are 8! state assignments to choose from. To keep things simple, we'll use the simplest, and assign the states in binary counting order, yielding the transition/excitation table in Table 7-15. Corresponding Karnaugh maps for D1, D2, and D3 are shown in Figure 7-58. The excitation equations can be read from the maps:

$$D1 = Q1 \cdot Q2' \cdot X + Q1' \cdot Q2 \cdot Q3 \cdot X' + Q1 \cdot Q2 \cdot Q3'$$

$$D2 = Q2' \cdot Q3 \cdot X + Q2 \cdot Q3' \cdot X$$

$$D3 = Q1 \cdot Q2' \cdot Q3' + Q1 \cdot Q3 \cdot X' + Q2' \cdot X' + Q3' \cdot Q1' \cdot X' + Q2 \cdot Q3' \cdot X$$

Figure 7-58 Excitation maps for D1, D2, and D3 in combination-lock machine.



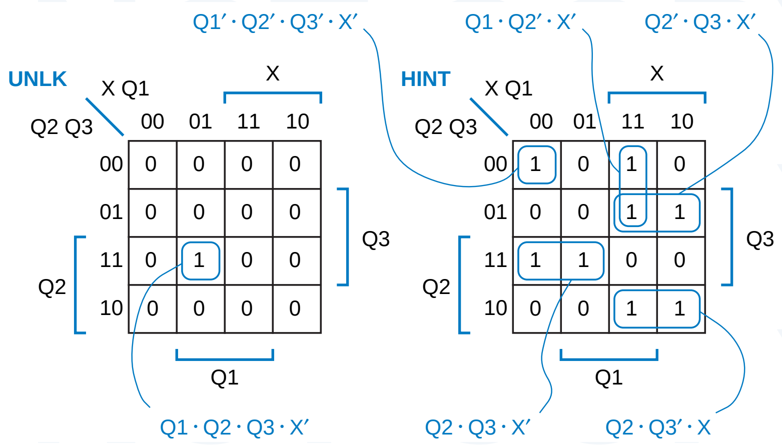


Figure 7-59 Karnaugh maps for output functions UNLK and HINT in combination-lock machine.

The output values are transferred from the transition/excitation and output table to another set of maps in Figure 7-59. The corresponding output equations are:

$$\text{UNLK} = \text{Q1} \cdot \text{Q2} \cdot \text{Q3} \cdot \text{X}'$$

$$\text{HINT} = \text{Q1}' \cdot \text{Q2}' \cdot \text{Q3}' \cdot \text{X}' + \text{Q1} \cdot \text{Q2}' \cdot \text{X} + \text{Q2}' \cdot \text{Q3} \cdot \text{X} + \text{Q2} \cdot \text{Q3} \cdot \text{X}' + \text{Q2} \cdot \text{Q3}' \cdot \text{X}$$

Note that some product terms are repeated in the excitation and output equations, yielding a slight savings in the cost of the AND-OR realization. If we went through the trouble of performing a formal multiple-output minimization of all five excitation and output functions, we could save two more gates (see Exercise 7.52).

7.5 Designing State Machines Using State Diagrams

Aside from planning the overall architecture of a digital system, designing state machines is probably the most creative task of a digital designer. Most people like to take a graphical approach to design—you've probably solved many problems just by doodling. For that reason, state diagrams are often used to design small- to medium-sized state machines. In this section, we'll give examples of state-diagram design, and describe a simple procedure for synthesizing circuits from the state diagrams. This procedure is the basis of the method used by CAD tools that can synthesize logic from graphical or even text-based "state diagrams."

Designing a state diagram is much like designing a state table, which, as we showed in Section 7.4.1, is much like writing a program. However, there is one fundamental difference between a state diagram and a state table, a difference that makes state-diagram design simpler but also more error prone:

- A state table is an exhaustive listing of the next states for each state/input combination. No ambiguity is possible.

- A state diagram contains a set of arcs labeled with transition expressions. Even when there are many inputs, only one transition expression is required per arc. However, when a state diagram is constructed, there is no guarantee that the transition expressions written on the arcs leaving a particular state cover all the input combinations exactly once.

ambiguous state diagram

In an improperly constructed (*ambiguous*) state diagram, the next state for some input combinations may be unspecified, which is generally undesirable, while multiple next states may be specified for others, which is just plain wrong. Thus, considerable care must be taken in the design of state diagrams; we'll give several examples.

Our first example is a state machine that controls the tail lights of a 1965 Ford Thunderbird, shown in Figure 7-60. There are three lights on each side, and for turns they operate in sequence to show the turning direction, as illustrated in Figure 7-61. The state machine has two input signals, LEFT and RIGHT, that indicate the driver's request for a left turn or a right turn. It also has an emergency-flasher input, HAZ, that requests the tail lights to be operated in hazard mode—all six lights flashing on and off in unison. We also assume the existence of a free-running clock signal whose frequency equals the desired flashing rate for the lights.

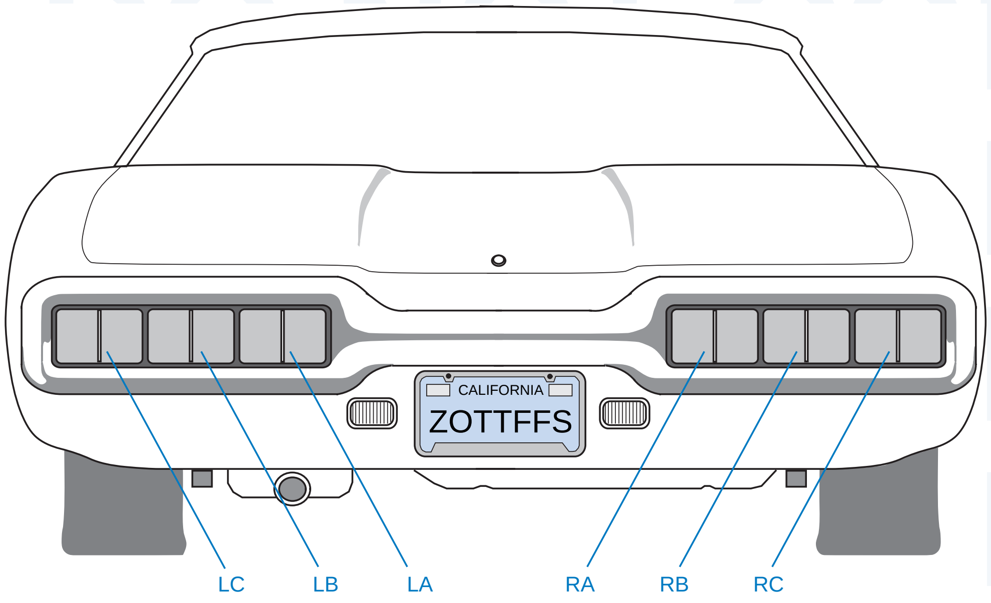


Figure 7-60
T-bird tail lights.

WHOSE REAR END? Actually, Figure 7-60 looks more like the rear end of a Mercury Capri, which also had sequential tail lights.

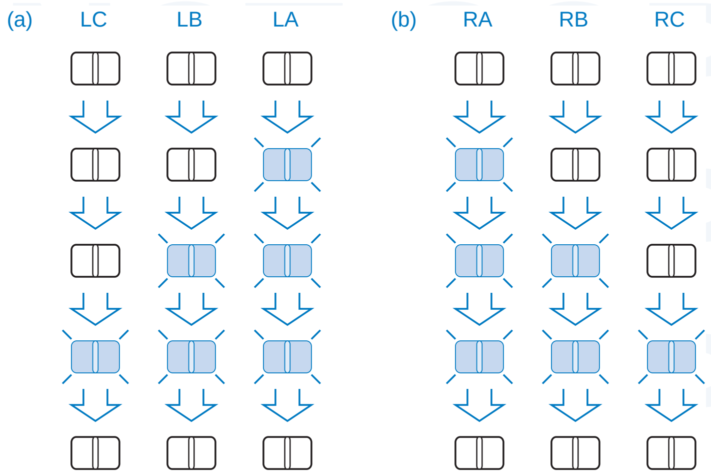
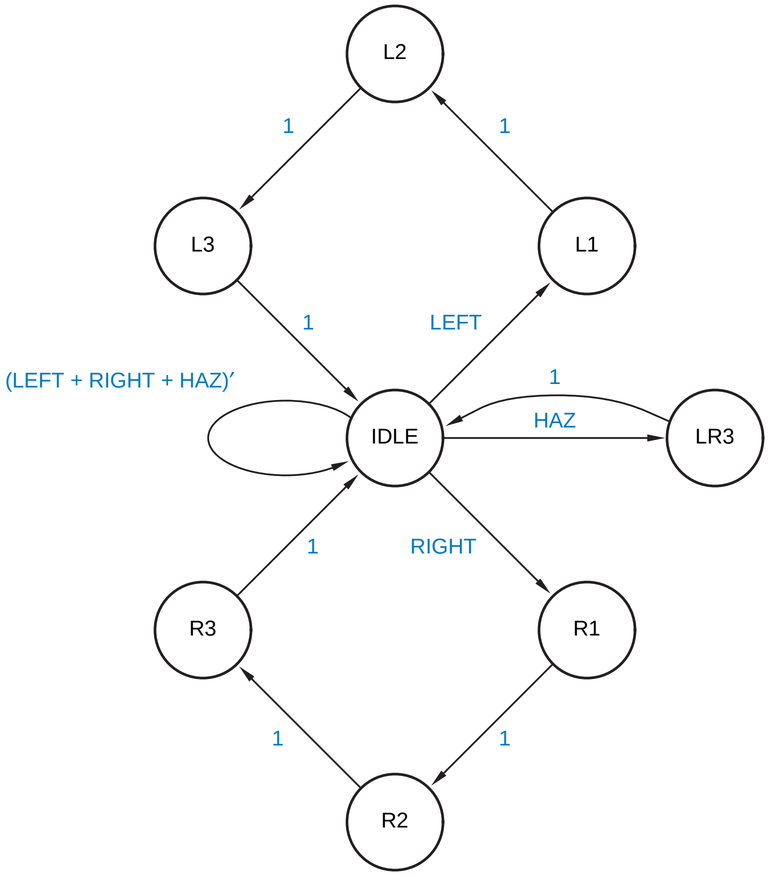


Figure 7-61
Flashing sequence for T-bird tail lights:
(a) left turn; (b) right turn.

Figure 7-62
Initial state diagram and output table for T-bird tail lights.



Output Table

State	LC	LB	LA	RA	RB	RC
IDLE	0	0	0	0	0	0
L1	0	0	1	0	0	0
L2	0	1	1	0	0	0
L3	1	1	1	0	0	0
R1	0	0	0	1	0	0
R2	0	0	0	1	1	0
R3	0	0	0	1	1	1
LR3	1	1	1	1	1	1

Given the foregoing requirements, we can design a clocked synchronous state machine to control the T-bird tail lights. We will design a Moore machine, so that the state alone determines which lights are on and which are off. For a left turn, the machine should cycle through four states, in which the right-hand lights are off and 0, 1, 2, or 3 of the left-hand lights are on. Likewise, for a right turn, it should cycle through four states in which the left-hand lights are off and 0, 1, 2, or 3 of the right-hand lights are on. In hazard mode, only two states are required—all lights on and all lights off.

Figure 7-62 shows our first cut at a state diagram for the machine. A common IDLE state is defined in which all of the lights are off. When a left turn is requested, the machine goes through three states in which 1, 2, and 3 of the left-hand lights are on, and then back to IDLE; right turns work similarly. In the hazard mode, the machine cycles back and forth between the IDLE state and a state in which all six lights are on. Since there are so many outputs, we've included a separate output table rather than writing output values on the state diagram. Even without assigning coded states to the named states, we can write output equations from the output table, if we let each state name represent a logic expression that is 1 only in that state:

$$
\begin{aligned}
\mathsf{LA} &= \mathsf{L1} + \mathsf{L2} + \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RA} &= \mathsf{R1} + \mathsf{R2} + \mathsf{R3} + \mathsf{LR3} \\
\mathsf{LB} &= \mathsf{L2} + \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RB} &= \mathsf{R2} + \mathsf{R3} + \mathsf{LR3} \\
\mathsf{LC} &= \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RC} &= \mathsf{R3} + \mathsf{LR3}
\end{aligned}
$$

There's one big problem with the state diagram of Figure 7-62—it doesn't properly handle multiple inputs asserted simultaneously. For example, what happens in the IDLE state if both LEFT and HAZ are asserted? According to the state diagram, the machine goes to two states, L1 and LR3, which is impossible. In reality, the machine would have only one next state, which could be L1, LR3, or a totally unrelated (and possibly unused) third state, depending on details of the state machine's realization (e.g., see Exercise 7.54).

The problem is fixed in Figure 7-63, where we have given the HAZ input priority. Also, we treat LEFT and RIGHT asserted simultaneously as a hazard request, since the driver is clearly confused and needs help.

The new state diagram is unambiguous because the transition expressions on the arcs leaving each state are mutually exclusive and all-inclusive. That is, for each state, no two expressions are 1 for the same input combination, and some expression is 1 for every input combination. This can be confirmed algebraically for this or any other state diagram by performing two steps:

mutual exclusion

1. *Mutual exclusion.* For each state, show that the logical product of each possible pair of transition expressions on arcs leaving that state is 0. If there are n arcs, then there are $n(n-1)/2$ logical products to evaluate.

all inclusion

2. *All inclusion.* For each state, show that the logical sum of the transition expressions on all arcs leaving that state is 1.

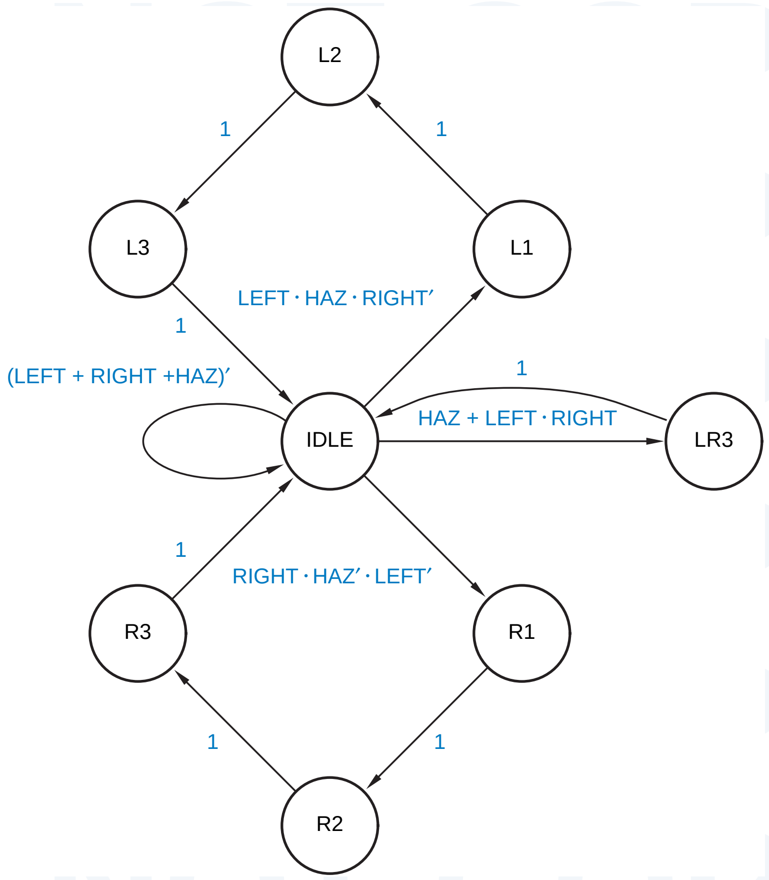


Figure 7-63
Corrected state diagram for T-bird tail lights.

If there are many transitions leaving each state, these steps, especially the first one, are very difficult to perform. However, typical state machines, even ones with lots of states and inputs, don't have many transitions leaving each state, since most designers can't dream up such complex machines in the first place. This is where the trade-off between state-table and state-diagram design occurs. In state-table design, the foregoing steps are not required, because the structure of a state table guarantees mutual exclusion and all inclusion. But if there are a lot of inputs, the state table has *lots* of columns.

Verifying that a state diagram is unambiguous may be difficult in principle, but it's not too bad in practice for small state diagrams. In Figure 7-63, most of the states have a single arc with a transition expression of 1, so verification is trivial. Real work is needed only to verify the IDLE state, which has four transitions leaving it. This can be done on a sheet of scratch paper by listing the eight combinations of the three inputs, and checking off the combinations covered by each transition expression. Each combination should have exactly one check. As another example, consider the state diagrams in Figures 7-44 and 7-46 on pages 462 and 464; both can be verified mentally.

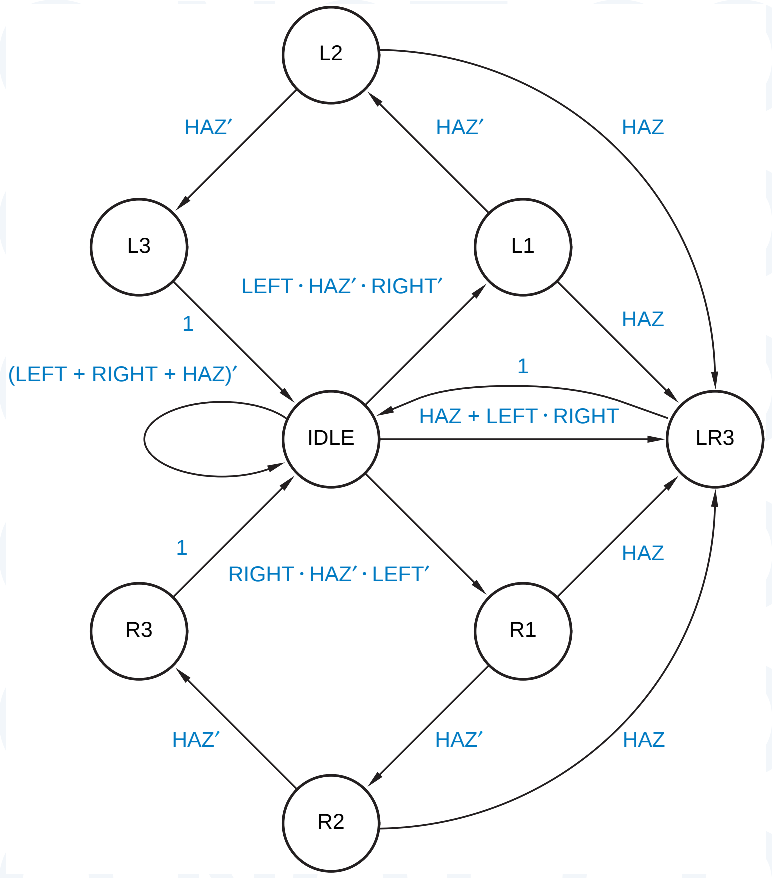


Figure 7-64 Enhanced state diagram for T-bird tail lights.

Returning to the T-bird tail lights machine, we can now synthesize a circuit from the state diagram if we wish. However, if we want to change the machine's behavior, now is the time to do it, before we do all the work of synthesizing a circuit. In particular, notice that once a left- or right-turn cycle has begun, the state diagram in Figure 7-63 allows the cycle to run to completion, even if HAZ is asserted. While this may have a certain aesthetic appeal, it would be safer for the car's occupants to have the machine go into hazard mode as soon as possible. The state diagram is modified to provide this behavior in Figure 7-64.

Now we're finally ready to synthesize a circuit for the T-bird machine. The state diagram has eight states, so we'll need a minimum of three flip-flops to code the states. Obviously, there are many state assignments possible (8! to be exact); we'll use the one in Table 7-16 for the following reasons:

1. An initial (idle) state of 000 is compatible with most flip-flops and registers, which are easily initialized to the 0 state.
2. Two state variables, Q1 and Q0, are used to "count" in Gray-code sequence for the left-turn cycle (IDLE→L1→L2→L3→IDLE). This minimizes the

Table 7-16
State assignment for T-bird tail lights state machine.

State	*Q2*	*Q1*	*Q0*
IDLE	0	0	0
L1	0	0	1
L2	0	1	1
L3	0	1	0
R1	1	0	1
R2	1	1	1
R3	1	1	0
LR3	1	0	0

number of state-variable changes per state transition, which can often simplify the excitation logic.

3. Because of the symmetry in the state diagram, the same sequence on Q1 and Q0 is used to “count” during a right-turn cycle, while Q2 is used to distinguish between left and right.
4. The remaining state-variable combination is used for the LR3 state.

The next step is to write a sort of transition table. However, we must use a format different from the transition tables of Section 7.4.4, because the transitions in a state diagram are specified by expressions rather than by an exhaustive tabulation of next states. We’ll call the new format a *transition list* because it has one row for each transition or arc in the state diagram.

transition list

Table 7-17 is the transition list for the state diagram of Figure 7-64 and the state assignment of Table 7-16. Each row contains the current state, next state, and transition expression for one arc in the state diagram. Both the named and coded versions of the current state and next state are shown. The named states are useful for reference purposes, while the coded states are used to develop transition equations.

Once we have a transition list, the rest of the synthesis steps are pretty much “turning the crank.” Synthesis procedures are described in Section 7.6. Although these procedures can be applied manually, they are usually embedded in a CAD software package; thus, Section 7.6 can help you understand what’s going on (or going wrong) in your favorite CAD package.

We also encountered one “turn-the-crank” step in this section—finding the ambiguities in state diagrams. Even though the procedure we discussed can be easily automated, few if any CAD programs perform this step in this way. For example, one “state diagram entry” tool silently removes duplicated transitions and goes to the state coded “00...00” for missing transitions, without warning the user. Thus, in most design environments, the designer is responsible for writing a state-machine description that is unambiguous. The state-machine description languages at the end of this chapter provide a good way to do this.

S	Q2	Q1	Q0	Transition expression	S*	Q2*	Q1*	Q0*
IDLE	0	0	0	(LEFT + RIGHT + HAZ)′	IDLE	0	0	0
IDLE	0	0	0	LEFT · HAZ′ · RIGHT′	L1	0	0	1
IDLE	0	0	0	HAZ + LEFT · RIGHT	LR3	1	0	0
IDLE	0	0	0	RIGHT · HAZ′ · LEFT′	R1	1	0	1
L1	0	0	1	HAZ′	L2	0	1	1
L1	0	0	1	HAZ	LR3	1	0	0
L2	0	1	1	HAZ′	L3	0	1	0
L2	0	1	1	HAZ	LR3	1	0	0
L3	0	1	0	1	IDLE	0	0	0
R1	1	0	1	HAZ′	R2	1	1	1
R1	1	0	1	HAZ	LR3	1	0	0
R2	1	1	1	HAZ′	R3	1	1	0
R2	1	1	1	HAZ	LR3	1	0	0
R3	1	1	0	1	IDLE	0	0	0
LR3	1	0	0	1	IDLE	0	0	0

Table 7-17 Transition list for T-bird tail lights state machine.

*7.6 State-Machine Synthesis Using Transition Lists

Once a machine's state diagram has been designed and a state assignment has been made, the creative part of the design process is pretty much over. The rest of the synthesis procedure can be carried out by CAD programs.

As we showed in the preceding section, a transition list can be constructed from a machine's state diagram and state assignment. This section shows how to synthesize a state machine from its transition list. It also delves into some of the options and nuances of state-machine design using transition lists. Although this material is useful for synthesizing machines by hand, its main purpose is to help you understand the internal operation and the external quirks of CAD programs and languages that deal with state machines.

*7.6.1 Transition Equations

The first step in synthesizing a state machine from a transition list is to develop a set of transition equations that define each next-state variable V* in terms of the current state and input. The transition list can be viewed as a sort of hybrid

* This section and all of its subsections are optional.

truth table in which the state-variable combinations for current-state are listed explicitly and input combinations are listed algebraically. Reading down a V∗ column in a transition list, we find a sequence of 0s and 1s, indicating the value of V∗ for various (if we've done it right, all) state/input combinations.

A transition equation for a next-state variable V∗ can be written using a sort of hybrid canonical sum:

$$V* = \sum_{\text{transition-list rows where } V* = 1} (\text{transition p-term})$$

transition p-term

That is, the transition equation has one "transition p-term" for each row of the transition list that contains a 1 in the V∗ column. A row's *transition p-term* is the product of the current state's minterm and the transition expression.

Based on the transition list in Table 7-17, the transition equation for Q2∗ in the T-bird machine can be written as the sum of eight p-terms:

$$\begin{aligned} Q2* = {} & Q2' \cdot Q1' \cdot Q0' \cdot (HAZ + LEFT \cdot RIGHT) \\ & + Q2' \cdot Q1' \cdot Q0' \cdot (RIGHT \cdot HAZ' \cdot LEFT') \\ & + Q2' \cdot Q1' \cdot Q0 \cdot (HAZ) \\ & + Q2' \cdot Q1 \cdot Q0 \cdot (HAZ) \\ & + Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ & + Q2 \cdot Q1' \cdot Q0 \cdot (HAZ) \\ & + Q2 \cdot Q1 \cdot Q0 \cdot (HAZ') \\ & + Q2 \cdot Q1 \cdot Q0 \cdot (HAZ) \end{aligned}$$

Some straightforward algebraic manipulations lead to a simplified transition equation that combines the first two, second two, and last four p-terms above:

$$\begin{aligned} Q2* = {} & Q2' \cdot Q1' \cdot Q0' \cdot (HAZ + RIGHT) \\ & + Q2' \cdot Q0 \cdot (HAZ) \\ & + Q2 \cdot Q0 \end{aligned}$$

Transition equations for Q1∗ and Q0∗ may be obtained in a similar manner:

$$\begin{aligned} Q1* = {} & Q2' \cdot Q1' \cdot Q0 \cdot (HAZ') \\ & + Q2' \cdot Q1 \cdot Q0 \cdot (HAZ') \\ & + Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ & + Q2 \cdot Q1 \cdot Q0 \cdot (HAZ') \\ = {} & Q0 \cdot HAZ' \\ Q0* = {} & Q2' \cdot Q1' \cdot Q0' \cdot (LEFT \cdot HAZ' \cdot RIGHT') \\ & + Q2' \cdot Q1' \cdot Q0' \cdot (RIGHT \cdot HAZ' \cdot LEFT') \\ & + Q2' \cdot Q1' \cdot Q0 \cdot (HAZ') \\ & + Q2 \cdot Q1' \cdot Q0 \cdot (HAZ') \\ = {} & Q2' \cdot Q1' \cdot Q0' \cdot HAZ' \cdot (LEFT \oplus RIGHT) + Q1' \cdot Q0 \cdot HAZ' \end{aligned}$$

Except for Q1∗, there's no guarantee that the transition equations above are in any sense minimal—in fact, the expressions for Q2∗ and Q0∗ aren't even in standard sum-of-products or product-of-sums form. The simplified equations, or the original unsimplified ones, merely provide an unambiguous starting point for whatever combinational design method you might choose to synthesize the excitation logic for the state machine—ad hoc, NAND-NAND, MSI-based, or whatever. In a PLD-based design, you could simply plug the equations into an ABEL program and let the compiler calculate the minimal sum-of-products expressions for the PLD's AND-OR array.

*7.6.2 Excitation Equations

While we're on the subject of excitation logic, note that so far we have derived only *transition equations*, not *excitation equations*. However, if we use D flip-flops as the memory elements in our state machines, then the excitation equations are trivial to derive from the transition equations, since the characteristic equation of a D flip-flop is $Q* = D$. Therefore, if the transition equation for a state variable Qi* is

$$Qi* = \text{expression}$$

then the excitation equation for the corresponding D flip-flop input is

$$Di = \text{expression}$$

Efficient excitation equations for other flip-flop types, especially J-K, are not so easy to derive (see Exercise 7.59). For that reason, the vast majority of discrete, PLD-based, and ASIC-based state-machine designs employ D flip-flops.

*7.6.3 Variations on the Scheme

There are other ways to obtain transition and excitation equations from a transition list. If the column for a particular next-state variable contains fewer 0s than 1s, it may be advantageous to write that variable's transition equation in terms of the 0s in its column. That is, we write

$$V*' = \sum_{\text{transition-list rows where } V* = 0} (\text{transition p-term})$$

That is, V*′ is 1 for all of the p-terms for which V* is 0. Thus, a transition equation for Q2*′ may be written as the sum of seven p-terms:

$$\begin{aligned}
Q2*' = {} & Q2' \cdot Q1' \cdot Q0' \cdot ((\text{LEFT} + \text{RIGHT} + \text{HAZ})') \\
& + Q2' \cdot Q1' \cdot Q0' \cdot (\text{LEFT} \cdot \text{HAZ}' \cdot \text{RIGHT}') \\
& + Q2' \cdot Q1' \cdot Q0 \cdot (\text{HAZ}') \\
& + Q2' \cdot Q1 \cdot Q0 \cdot (\text{HAZ}') \\
& + Q2' \cdot Q1 \cdot Q0' \cdot (1) \\
& + Q2 \cdot Q1 \cdot Q0' \cdot (1) \\
& + Q2 \cdot Q1' \cdot Q0' \cdot (1) \\
= {} & Q2' \cdot Q1' \cdot Q0' \cdot \text{HAZ}' \cdot \text{RIGHT}' + Q2' \cdot Q0 \cdot \text{HAZ}' + Q1 \cdot Q0' + Q2 \cdot Q0'
\end{aligned}$$

To obtain an equation for Q2∗, we simply complement both sides of the reduced equation.

To obtain an expression for a next-state variable V∗ directly using the 0s in the transition list, we can complement the right-hand side of the general V∗′ equation using DeMorgan's theorem, obtaining a sort of hybrid canonical product:

$$\mathrm{V*} = \prod_{\text{transition-list rows where } \mathrm{V*} = 0} (\text{transition s-term})$$

transition s-term

Here, a row's *transition s-term* is the sum of the maxterm for the current state and the complement of the transition expression. If the transition expression is a simple product term, then its complement is a sum, and the transition equation expresses V∗ in product-of-sums form.

*7.6.4 Realizing the State Machine

Once you have the excitation equations for a state machine, all you're left with is a multiple-output combinational logic design problem. Of course, there are many ways to realize combinational logic from equations, but the easiest way is just to type them into an ABEL or VHDL program and use the compiler to synthesize a PLD, FPGA, or ASIC realization.

Combinational PLDs such as the PAL16L8 and GAL16V8 that we studied in Section 5.3 can be used to realize excitation equations up to a certain number of inputs, outputs, and product terms. Better yet, in Section 8.3 we'll introduce sequential PLDs that include D flip-flops on the same chip with the combinational AND-OR array. For a given number of PLD input and output pins, these sequential PLDs can realize larger state machines than their combinational counterparts, because the excitation signals never have to go off the chip. In \secref{PLDtranlist}, we'll show how to realize the T-bird tail-lights excitation equations in a sequential PLD.

*7.7 Another State-Machine Design Example

This section gives one more example of state-machine design using a state diagram. The example provides a basis for further discussion of a few topics: unused states, output-coded state assignments, and "don't-care" state codings.

*7.7.1 The Guessing Game

Our final state-machine example is a "guessing game" that can be built as an amusing lab project:

Design a clocked synchronous state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to lamps or LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red lamp. In normal operation, the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.

Guesses are made by pressing a pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the "wrong" pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the lamp output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

Clearly, we will have to provide four states, one for each position of the rotating pattern, and we'll need at least one state to indicate that play has stopped. A possible state diagram is shown in Figure 7-65. The machine cycles through states S1–S4 as long as no Gi input is asserted, and it goes to the STOP state when a guess is made. Each Li output is asserted in the like-numbered state.

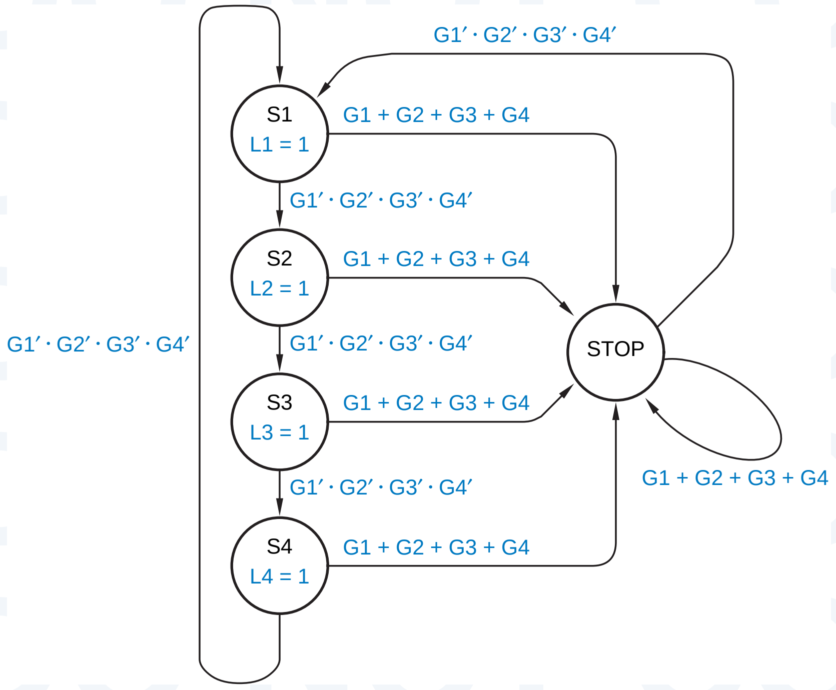


Figure 7-65 First try at a state diagram for the guessing game.

The only problem with this state diagram is that it doesn't "remember" in the STOP state whether the guess was correct, so it has no way to control the ERR output. This problem is fixed in Figure 7-66, which has two "stopped" states, SOK and SERR. On an incorrect guess, the machine goes to SERR, where ERR is asserted; otherwise, it goes to SOK. Although the machine's word description doesn't require it, the state diagram is designed to go to SERR even if the user tries to fool it by pressing two or more pushbuttons simultaneously, or by changing guesses while stopped.

A transition list corresponding to the state diagram is shown in Table 7-18, using a simple 3-bit binary state encoding with Gray-code order for the S1–S4 cycle. Transition equations for Q1∗ and Q0∗ can be obtained from the table as follows:

$$\begin{aligned}
Q1* &= Q2' \cdot Q1' \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1 \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&= Q2' \cdot Q0 \cdot G1' \cdot G2' \cdot G3' \cdot G4' \\
Q0* &= Q2' \cdot Q1' \cdot Q0' \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1' \cdot Q0' \cdot (G2 + G3 + G4) \\
&\quad + Q2' \cdot Q1' \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1' \cdot Q0 \cdot (G1 + G3 + G4) \\
&\quad + Q2' \cdot Q1 \cdot Q0 \cdot (G1 + G2 + G4) \\
&\quad + Q2' \cdot Q1 \cdot Q0' \cdot (G1 + G2 + G3) \\
&\quad + Q2 \cdot Q1' \cdot Q0 \cdot (G1 + G2 + G3 + G4)
\end{aligned}$$

Using a logic minimization program, the Q0∗ expression can be reduced to 11 product terms in two-level sum-of-products form. An expression for Q2∗ is best formulated in terms of the 0s in the Q2∗ column of the transition list:

$$\begin{aligned}
Q2*' &= Q2' \cdot Q1' \cdot Q0' \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1' \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1 \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2' \cdot Q1 \cdot Q0' \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2 \cdot Q1' \cdot Q0' \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&\quad + Q2 \cdot Q1' \cdot Q0 \cdot (G1' \cdot G2' \cdot G3' \cdot G4') \\
&= (Q2' + Q1') \cdot (G1' \cdot G2' \cdot G3' \cdot G4')
\end{aligned}$$

The last five columns of Table 7-18 show output values. Thus, output equations can be developed in much the same way as transition equations. However, since this example is a Moore machine, outputs are independent of the transition expressions; only one row of the transition list must be considered for each current state. The output equations are

$$\begin{aligned}
L1 &= Q2' \cdot Q1' \cdot Q0' & L3 &= Q2' \cdot Q1 \cdot Q0 & ERR &= Q2 \cdot Q1' \cdot Q0 \\
L2 &= Q2' \cdot Q1' \cdot Q0 & L4 &= Q2' \cdot Q1 \cdot Q0' & &
\end{aligned}$$

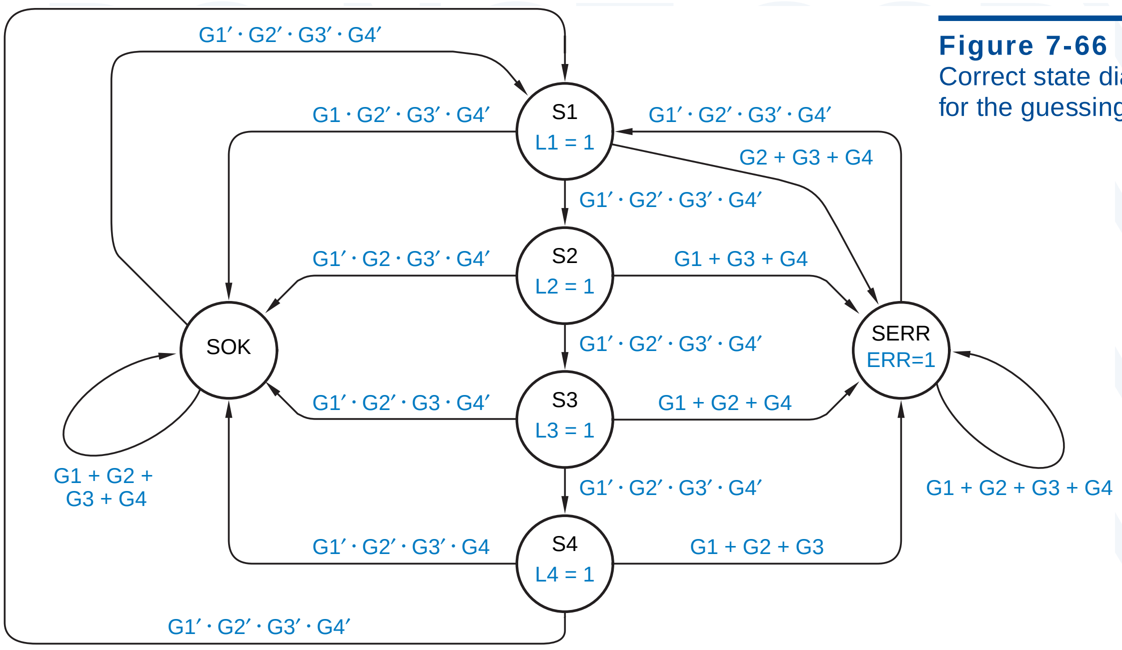


Figure 7-66 Correct state diagram for the guessing game.

Table 7-18 Transition list for guessing-game machine.

Current State					*Next State*				*Output*				
S	***Q2***	***Q1***	***Q0***	***Transition Expression***	***S****	***Q2****	***Q1****	***Q0****	***L1***	***L2***	***L3***	***L4***	***ERR***
S1	0	0	0	G1′ · G2′ · G3′ · G4′	S2	0	0	1	1	0	0	0	0
S1	0	0	0	G1 · G2′ · G3′ · G4′	SOK	1	0	0	1	0	0	0	0
S1	0	0	0	G2 + G3 + G4	SERR	1	0	1	1	0	0	0	0
S2	0	0	1	G1′ · G2′ · G3′ · G4′	S3	0	1	1	0	1	0	0	0
S2	0	0	1	G1′ · G2 · G3′ · G4′	SOK	1	0	0	0	1	0	0	0
S2	0	0	1	G1 + G3 + G4	SERR	1	0	1	0	1	0	0	0
S3	0	1	1	G1′ · G2′ · G3′ · G4′	S4	0	1	0	0	0	1	0	0
S3	0	1	1	G1′ · G2′ · G3 · G4′	SOK	1	0	0	0	0	1	0	0
S3	0	1	1	G1 + G2 + G4	SERR	1	0	1	0	0	1	0	0
S4	0	1	0	G1′ · G2′ · G3′ · G4′	S1	0	0	0	0	0	0	1	0
S4	0	1	0	G1′ · G2′ · G3′ · G4	SOK	1	0	0	0	0	0	1	0
S4	0	1	0	G1 + G2 + G3	SERR	1	0	1	0	0	0	1	0
SOK	1	0	0	G1 + G2 + G3 + G4	SOK	1	0	0	0	0	0	0	0
SOK	1	0	0	G1′ · G2′ · G3′ · G4′	S1	0	0	0	0	0	0	0	0
SERR	1	0	1	G1 + G2 + G3 + G4	SERR	1	0	1	0	0	0	0	1
SERR	1	0	1	G1′ · G2′ · G3′ · G4′	S1	0	0	0	0	0	0	0	1

*7.7.2 Unused States

Our state diagram for the guessing game has six states, but the actual state machine, built from three flip-flops, has eight. By omitting the unused states from the transition list, we treated them as "don't-cares" in a very limited sense:

- When we wrote equations for Q1∗ and Q0∗, we formed a sum of transition p-terms for state/input combinations that had an explicit 1 in the corresponding columns of the transition list. Although we didn't consider the unused states, our procedure implicitly treated them as if they had 0s in the Q1∗ and Q0∗ columns.
- Conversely, we wrote the Q2∗′ equation as a sum of transition p-terms for state/input combinations that had an explicit 0 in the corresponding columns of the transition list. Unused states were implicitly treated as if they had 1s in the Q2∗ column.

As a consequence of these choices, all of the unused states in the guessing-game machine have a coded next state of 100 for all input combinations. That's safe, acceptable behavior should the machine stray into an unused state, since 100 is the coding for one of the normal states (SOK).

To treat the unused states as true "don't-cares," we would have to allow them to go to any next state under any input combination. This is simple in principle but may be difficult in practice.

At the end of Section 7.4.4, we showed how to handle unused states as "don't-cares" in the Karnaugh-map method for developing transition/excitation equations. Unfortunately, for all but the smallest problems, Karnaugh maps are unwieldy. Commercially available logic minimization programs can easily handle larger problems, but many of them don't handle "don't-cares" or require the designer to insert special code to handle them. In ABEL state machines, don't-care next states can be handled fairly easily using the @DCSET directive, as we discuss in the box on page 534. In VHDL, the process is a bit unwieldy.

*7.7.3 Output-Coded State Assignment

Let's look at another realization of the guessing-game machine. The machine's outputs are a function of state only; furthermore, a *different* output combination is produced in each named state. Therefore, we can use the outputs as state variables and assign each named state to the required output combination. This sort of *output-coded state assignment* can sometimes result in excitation equations that are simpler than the set of excitation and output equations obtained with a state assignment using a minimum number of state variables.

output-coded state assignment

Table 7-19 is the guessing-game transition list that results from an output-coded state assignment. Each transition/excitation equation has very few transition p-terms because the transition list has so few 1s in the next-state.columns:

Table 7-19
Transition list for guessing-game machine using outputs as state variables.

Current State							Next State					
S	***L1***	***L2***	***L3***	***L4***	***ERR***	***Transition Expression***	***S****	***L1****	***L2****	***L3****	***L4****	***ERR****
S1	1	0	0	0	0	G1′ · G2′ · G3′ · G4′	S2	0	1	0	0	0
S1	1	0	0	0	0	G1 · G2′ · G3′ · G4′	SOK	0	0	0	0	0
S1	1	0	0	0	0	G2 + G3 + G4	SERR	0	0	0	0	1
S2	0	1	0	0	0	G1′ · G2′ · G3′ · G4′	S3	0	0	1	0	0
S2	0	1	0	0	0	G1′ · G2 · G3′ · G4′	SOK	0	0	0	0	0
S2	0	1	0	0	0	G1 + G3 + G4	SERR	0	0	0	0	1
S3	0	0	1	0	0	G1′ · G2′ · G3′ · G4′	S4	0	0	0	1	0
S3	0	0	1	0	0	G1′ · G2′ · G3 · G4′	SOK	0	0	0	0	0
S3	0	0	1	0	0	G1 + G2 + G4	SERR	0	0	0	0	1
S4	0	0	0	1	0	G1′ · G2′ · G3′ · G4′	S1	1	0	0	0	0
S4	0	0	0	1	0	G1′ · G2′ · G3′ · G4	SOK	0	0	0	0	0
S4	0	0	0	1	0	G1 + G2 + G3	SERR	0	0	0	0	1
SOK	0	0	0	0	0	G1 + G2 + G3 + G4	SOK	0	0	0	0	0
SOK	0	0	0	0	0	G1′ · G2′ · G3′ · G4′	S1	1	0	0	0	0
SERR	0	0	0	0	1	G1 + G2 + G3 + G4	SERR	0	0	0	0	1
SERR	0	0	0	0	1	G1′ · G2′ · G3′ · G4′	S1	1	0	0	0	0

$$
\begin{aligned}
\text{L1}* &= \text{L1}' \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4} \cdot \text{ERR}' \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
&\quad + \text{L1}' \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
&\quad + \text{L1}' \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR} \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
\text{L2}* &= \text{L1} \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
\text{L3}* &= \text{L1}' \cdot \text{L2} \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
\text{L4}* &= \text{L1}' \cdot \text{L2}' \cdot \text{L3} \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1}' \cdot \text{G2}' \cdot \text{G3}' \cdot \text{G4}') \\
\text{ERR}* &= \text{L1} \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G2} + \text{G3} + \text{G4}) \\
&\quad + \text{L1}' \cdot \text{L2} \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1} + \text{G3} + \text{G4}) \\
&\quad + \text{L1}' \cdot \text{L2}' \cdot \text{L3} \cdot \text{L4}' \cdot \text{ERR}' \cdot (\text{G1} + \text{G2} + \text{G4}) \\
&\quad + \text{L1}' \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4} \cdot \text{ERR}' \cdot (\text{G1} + \text{G2} + \text{G3}) \\
&\quad + \text{L1}' \cdot \text{L2}' \cdot \text{L3}' \cdot \text{L4}' \cdot \text{ERR} \cdot (\text{G1} + \text{G2} + \text{G3} + \text{G4})
\end{aligned}
$$

There are no output equations, of course. The ERR* equation above is the worst in the group, requiring 16 terms to express in either minimal sum-of-products or product-of-sums form.

As a group, the equations developed above have just about the same complexity as the transition and output equations that we developed from Table 7-18. Even though the output-coded assignment does not produce a simpler set of equations in this example, it can still save cost in a PLD-based design, since fewer PLD macrocells or outputs are needed overall.

*7.7.4 “Don’t-Care” State Codings

Out of the 32 possible coded states using five variables, only six are used in Table 7-19. The rest of the states are unused and have a next state of 00000 if the machine is built using the equations in the preceding subsection. Another possible disposition for unused states, one that we haven’t discussed before, is obtained by careful use of “don’t-cares” in the assignment of coded states to current states.

Table 7-20 shows one such state assignment for the guessing-game machine, derived from the output-coded state assignment of the preceding subsection. In this example, every possible combination of current-state variables corresponds to one of the coded states (e.g., 10111 = S1, 00101 = S3). However, *next states* are coded using the same unique combinations as in the preceding subsection. Table 7-21 shows the resulting transition list.

In this approach, each unused current state behaves like a nearby “normal” state; Figure 7-67 illustrates the concept. The machine is well-behaved and goes to a “normal state” if it inadvertently enters an unused state. Yet the approach still allows some simplification of the excitation and output logic by introducing

Table 7-20 Current-state assignment for the guessing-game machine using don’t-cares.

State	*L1*	*L2*	*L3*	*L4*	*ERR*
S1	1	x	x	x	x
S2	0	1	x	x	x
S3	0	0	1	x	x
S4	0	0	0	1	x
SOK	0	0	0	0	0
SERR	0	0	0	0	1

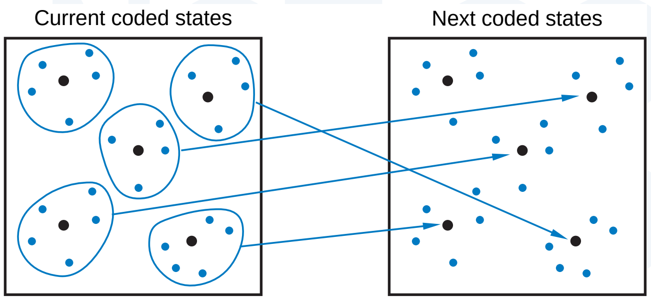


Figure 7-67 State assignment using don’t-cares for current states.

Table 7-21
Transition list for guessing-game machine using don't-care state codings.

Current State							Next State					
S	***L1***	***L2***	***L3***	***L4***	***ERR***	***Transition Expression***	***S****	***L1****	***L2****	***L3****	***L4****	***ERR****
S	1	x	x	x	x	$G1' \cdot G2' \cdot G3' \cdot G4'$	S2	0	1	0	0	0
S	1	x	x	x	x	$G1 \cdot G2' \cdot G3' \cdot G4'$	SOK	0	0	0	0	0
S	1	x	x	x	x	$G2 + G3 + G4$	SERR	0	0	0	0	1
S2	0	1	x	x	x	$G1' \cdot G2' \cdot G3' \cdot G4'$	S3	0	0	1	0	0
S2	0	1	x	x	x	$G1' \cdot G2 \cdot G3' \cdot G4'$	SOK	0	0	0	0	0
S2	0	1	x	x	x	$G1 + G3 + G4$	SERR	0	0	0	0	1
S3	0	0	1	x	x	$G1' \cdot G2' \cdot G3' \cdot G4'$	S4	0	0	0	1	0
S3	0	0	1	x	x	$G1' \cdot G2' \cdot G3 \cdot G4'$	SOK	0	0	0	0	0
S3	0	0	1	x	x	$G1 + G2 + G4$	SERR	0	0	0	0	1
S4	0	0	0	1	x	$G1' \cdot G2' \cdot G3' \cdot G4'$	S1	1	0	0	0	0
S4	0	0	0	1	x	$G1' \cdot G2' \cdot G3' \cdot G4$	SOK	0	0	0	0	0
S4	0	0	0	1	x	$G1 + G2 + G3$	SERR	0	0	0	0	1
SOK	0	0	0	0	0	$G1 + G2 + G3 + G4$	SOK	0	0	0	0	0
SOK	0	0	0	0	0	$G1' \cdot G2' \cdot G3' \cdot G4'$	S1	1	0	0	0	0
SERR	0	0	0	0	1	$G1 + G2 + G3 + G4$	SERR	0	0	0	0	1
SERR	0	0	0	0	1	$G1' \cdot G2' \cdot G3' \cdot G4'$	S1	1	0	0	0	0

don't-cares in the transition list. When a row's transition p-term is written, current-state variables that are don't-cares in that row are omitted, for example,

$$\begin{aligned} ERR* = {} & L1 \cdot (G2 + G3 + G4) \\ & + L1' \cdot L2 \cdot (G1 + G3 + G4) \\ & + L1' \cdot L2' \cdot L3 \cdot (G1 + G2 + G4) \\ & + L1' \cdot L2' \cdot L3' \cdot L4 \cdot (G1 + G2 + G3) \\ & + L1' \cdot L2' \cdot L3' \cdot L4' \cdot ERR \cdot (G1 + G2 + G3 + G4) \end{aligned}$$

Compared with the ERR* equation in the preceding subsection, the one above still requires 16 terms to express as a sum of products. However, it requires only five terms in minimal product-of-sums form, which makes its complement more suitable for realization in a PLD.

*7.8 Decomposing State Machines

Just like large procedures or subroutines in a programming language, large state machines are difficult to conceptualize, design, and debug. Therefore, when faced with a large state-machine problem, digital designers often look for opportunities to solve it with a collection of smaller state machines.

state-machine decomposition

There's a well-developed theory of *state-machine decomposition* that you can use to analyze any given, monolithic state machine to determine whether it can be realized as a collection of smaller ones. However, decomposition theory is not too useful for designers who want to avoid designing large state machines in the first place. Rather, a practical designer tries to cast the original design problem into a natural, hierarchical structure, so that the uses and functions of submachines are obvious, making it unnecessary ever to write a state table for the equivalent monolithic machine.

main machine
submachines

The simplest and most commonly used type of decomposition is illustrated in Figure 7-68. A *main machine* provides the primary inputs and outputs and executes the top-level control algorithm. *Submachines* perform low-level steps under the control of the main machine, and may optionally handle some of the primary inputs and outputs.

Perhaps the most commonly used submachine is a counter. The main machine starts the counter when it wishes to stay in a particular main state for *n* clock ticks, and the counter asserts a DONE signal when *n* ticks have occurred. The main machine is designed to wait in the same state until DONE is asserted. This adds an extra output and input to the main machine (START and DONE), but it saves *n* – 1 states.

An example decomposed state machine designed along these lines is based on the guessing game of Section 7.7. The original guessing game is easy to win after a minute of practice because the lamps cycle at a very consistent rate of

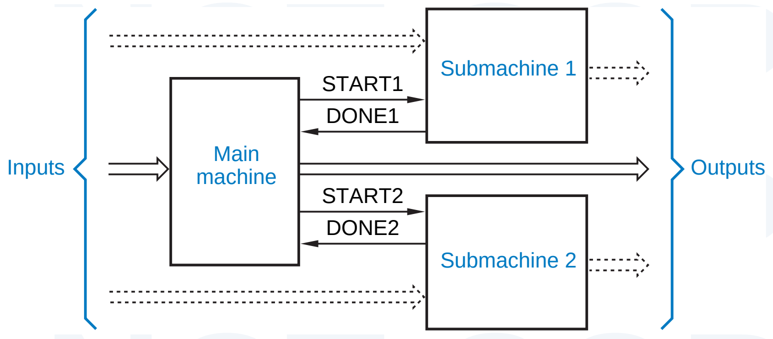


Figure 7-68
A typical, hierarchical state-machine structure.

A REALLY BAD JOKE Note that the title of this section has nothing to do with the "buried flip-flops" found in some PLDs.

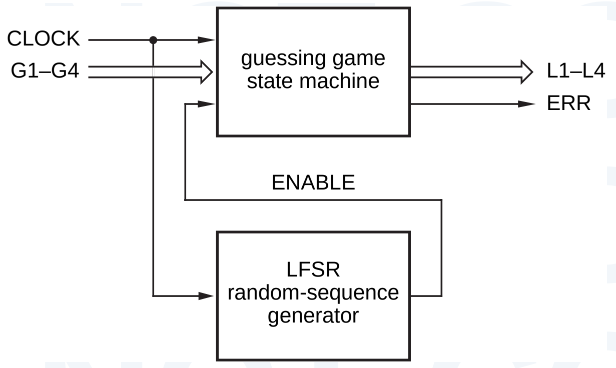


Figure 7-69
Block diagram of guessing game with random delay.

4 Hz. To make the game more challenging, we can double or triple the clock speed, but allow the lamps to stay in each state for a random length of time. Then the user truly must guess whether a given lamp will stay on long enough for the corresponding pushbutton to be pressed.

A block diagram for the enhanced guessing game is shown in Figure 7-69. The main machine is basically the same as before, except that it only advances from one lamp state to the next if the enable input EN is asserted, as shown by the state diagram in Figure 7-70. The enable input is driven by the output of a pseudo-random sequence generator, a linear feedback shift register (LFSR).

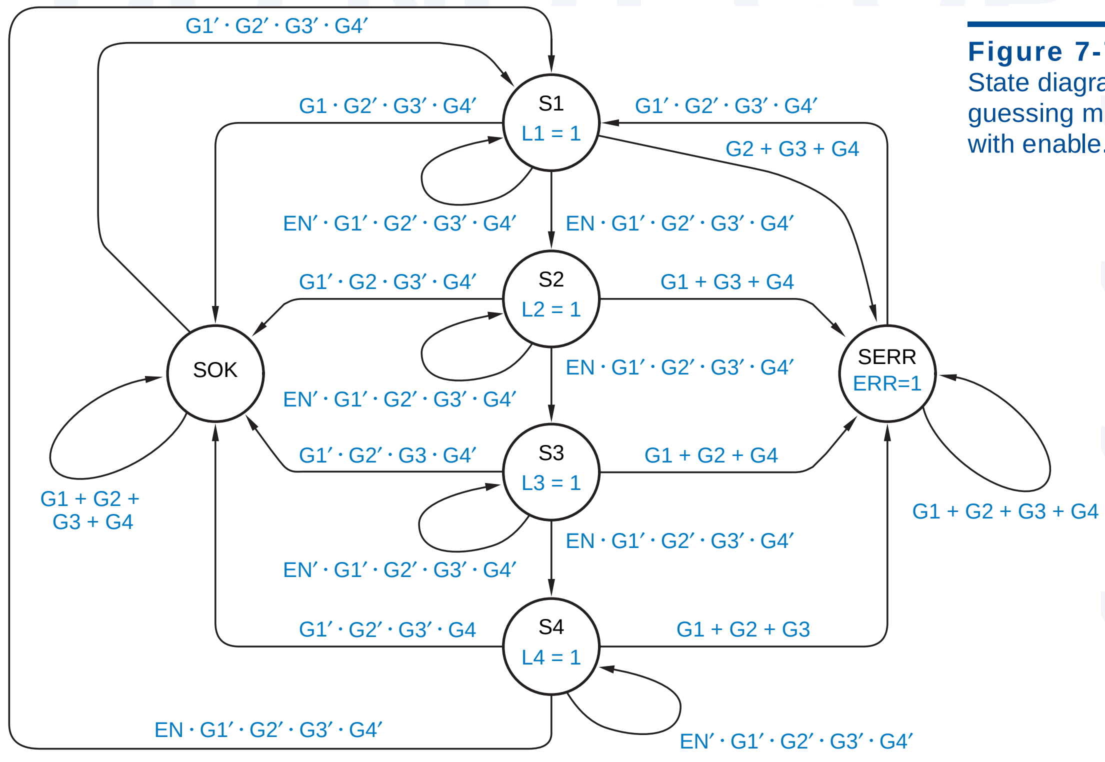


Figure 7-70
State diagram for guessing machine with enable.

A SHIFTY CIRCUIT

LFSR circuits are described in \secref{LFSRctrs}. In Figure 7-69, the low-order bit of an n-bit LFSR counter is used as the enable signal. Thus, the length of time that a particular lamp stays on depends on the counting sequence of the LFSR.

In the best case for the user, the LFSR contains 10…00; in this case the lamp is on for $n-1$ clock ticks because it takes that long for the single 1 to shift into the low-order bit position. In the worst case, the LFSR contains 11…11 and shifting occurs for n consecutive clock ticks. At other times, shifting stops for a time determined by the number of consecutive 0s starting with the low-order bit of the LFSR.

All of these cases are quite unpredictable unless the user has memorized the shifting cycle (2^n-1 states) or is very fast at Galois-field arithmetic. Obviously, a large value of n (≥ 16) provides the most fun.

Another obvious candidate for decomposition is a state machine that performs binary multiplication using the shift-and-add algorithm, or binary division using the shift-and-subtract algorithm. To perform an n-bit operation, these algorithms require an initialization step, n computation steps, and possible cleanup steps. The main machine for such an algorithm contains states for initialization, generic computation, and cleanup steps, and a modulo-n counter can be used as a submachine to control the number of generic computation steps executed.

*7.9 Feedback Sequential Circuits

The simple bistable and the various latches and flip-flops that we studied earlier in this chapter are all feedback sequential circuits. Each has one or more feedback loops that, ignoring their behavior during state transitions, store a 0 or a 1 at all times. The feedback loops are memory elements, and the circuits' behavior depends on both the current inputs and the values stored in the loops.

*7.9.1 Analysis

fundamental-mode circuit

Feedback sequential circuits are the most common example of *fundamental-mode circuits*. In such circuits, inputs are not normally allowed to change simultaneously. The analysis procedure assumes that inputs change one at a time, allowing enough time between successive changes for the circuit to settle into a stable internal state. This differs from clocked circuits, in which multiple inputs can change at almost arbitrary times without affecting the state, and all input values are sampled and state changes occur with respect to a clock signal.

*This section and all of its subsections are optional.

KEEP YOUR FEEDBACK TO YOURSELF

Only rarely does a logic designer encounter a situation where a feedback sequential circuit must be analyzed or designed. The most commonly used feedback sequential circuits are the flip-flops and latches that are used as the building blocks in larger sequential-circuit designs. Their internal design and operating specifications are supplied by an IC manufacturer.

Even an ASIC designer typically does not design gate-level flip-flop or latch circuits, since these elements are supplied in a "library" of commonly used functions in the particular ASIC technology. Still, you may be curious about how off-the-shelf flip-flops and latches "do their thing"; this section will show you how to analyze such circuits.

Like clocked synchronous state machines, feedback sequential circuits may be structured as Mealy or Moore circuits, as shown in Figures 7-71. A circuit with *n* feedback loops has *n* binary state variables and 2^n states.

To analyze a feedback sequential circuit, we must break the feedback loops in Figure 7-71 so that the next value stored in each loop can be predicted as a function of the circuit inputs and the current value stored in all loops. Figure 7-72 shows how to do this for the NAND circuit for a D latch, which has only one feedback loop. We conceptually break the loop by inserting a fictional buffer in the loop as shown. The output of the buffer, named Y, is the single state variable for this example.

Let us assume that the propagation delay of the fictional buffer is 10 ns (but any nonzero number will do), and that all of the other circuit components have zero delay. If we know the circuit's current state (Y) and inputs (D and C), then we can predict the value Y will have in 10 ns. The next value of Y, denoted Y∗, is a combinational function of the current state and inputs. Thus, reading the circuit diagram, we can write an *excitation equation* for Y∗:

excitation equation

$$\begin{aligned} Y* &= (C \cdot D) + (C \cdot D' + Y')' \\ &= C \cdot D + C' \cdot Y + D \cdot Y \end{aligned}$$

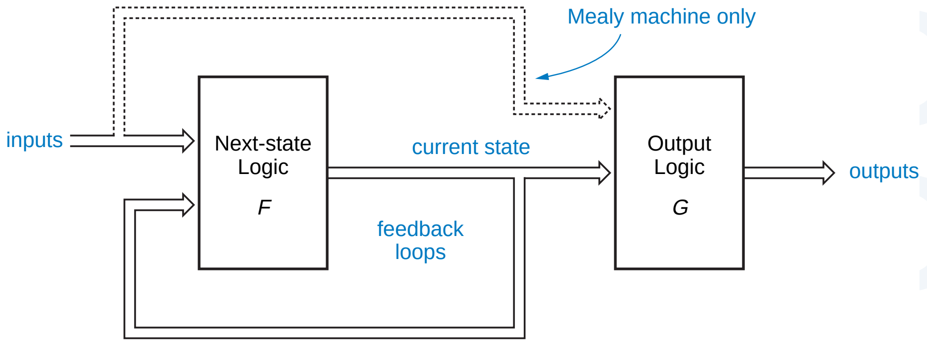


Figure 7-71 Feedback sequential circuit structure for Mealy and Moore machines.

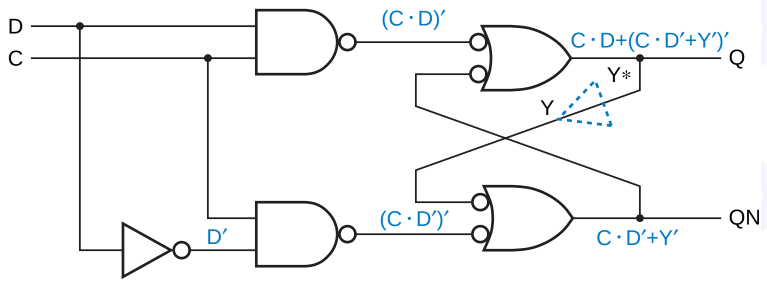


Figure 7-72
Feedback analysis of a D latch.

transition table

Now the state of the feedback loop (and the circuit) can be written as a function of the current state and input, and enumerated by a *transition table* as shown in Figure 7-73. Each cell in the transition table shows the fictional-buffer output value that will occur 10 ns (or whatever delay you've assumed) after the corresponding state and input combination occurs.

Figure 7-73
Transition table for the D latch in Figure 7-72.

	C D			
Y	00	01	11	10
0	0	0	1	0
1	1	1	1	0
	Y∗			

A transition table has one row for each possible combination of the state variables, so a circuit with n feedback loops has 2^n rows in its transition table. The table has one column for each possible input combination, so a circuit with m inputs has 2^m columns in its transition table.

By definition, a fundamental-mode circuit such as a feedback sequential circuit does not have a clock to tell it when to sample its inputs. Instead, we can imagine that the circuit is evaluating its current state and input *continuously* (or every 10 ns, if you prefer). As the result of each evaluation, it goes to a next state predicted by the transition table. Most of the time, the next state is the same as the current state; this is the essence of fundamental-mode operation. We make some definitions below that will help us study this behavior in more detail.

JUST ONE LOOP

The way the circuit in Figure 7-72 is drawn, it may look like there are two feedback loops. However, once we make one break as shown, there are no more loops. That is, each signal can be written as a combinational function of the other signals, not including itself.

	C D			
S	00	01	11	10
S0	$\textcircled{\text{S0}}$	$\textcircled{\text{S0}}$	S1	$\textcircled{\text{S0}}$
S1	$\textcircled{\text{S1}}$	$\textcircled{\text{S1}}$	$\textcircled{\text{S1}}$	S0
	S*			

Figure 7-74
State table for the D latch in Figure 7-72. showing stable total states.

In a fundamental-mode circuit, a *total state* is a particular combination of *internal state* (the values stored in the feedback loops) and *input state* (the current value of the circuit inputs). A *stable total state* is a combination of internal state and input state such that the next internal state predicted by the transition table is the same as the current internal state. If the next internal state is different, then the combination is an *unstable total state*. We have rewritten the transition table for the D latch in Figure 7-74 as a *state table*, giving the names S0 and S1 to the states and drawing a circle around the stable total states.

total state
internal state
input state
stable total state
unstable total state
state table

To complete the analysis of the circuit, we must also determine how the outputs behave as functions of the internal state and inputs. There are two outputs, and hence two *output equations*:

output equation

$$
\begin{aligned}
Q &= C \cdot D + C' \cdot Y + D \cdot Y \\
QN &= C \cdot D' + Y'
\end{aligned}
$$

Note that Q and QN are *outputs*, not state variables. Even though the circuit has two outputs, which can theoretically take on four combinations, it has only one state variable Y, and hence only two states.

The output values predicted by the Q and QN equations can be incorporated in a combined state and output table that completely describes the operation of the circuit, as shown in Figure 7-75. Although Q and QN are normally complementary, it is possible for them to have the same value (1) momentarily, during the transition from S0 to S1 under the C D = 11 column of the table.

We can now predict the behavior of the circuit from the transition and output table. First of all, notice that we have written the column labels in our state tables in “Karnaugh map” order, so that only a single input bit changes

	C D			
S	00	01	11	10
S0	$\textcircled{\text{S0}}$, 01	$\textcircled{\text{S0}}$, 01	S1 , 11	$\textcircled{\text{S0}}$, 01
S1	$\textcircled{\text{S1}}$, 10	$\textcircled{\text{S1}}$, 10	$\textcircled{\text{S1}}$, 10	S0 , 01
	S*, Q QN			

Figure 7-75
State and output table for the D latch.

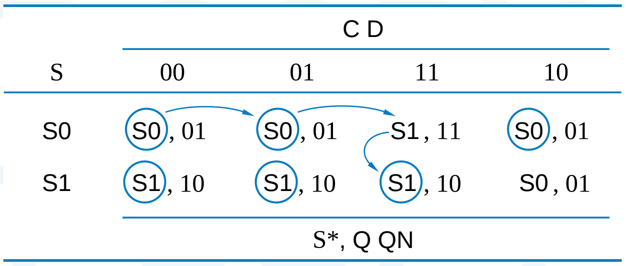


Figure 7-76
Analysis of the D latch for a few transitions.

S	CD = 00	01	11	10
S0	S0, 01	S0, 01	S1, 11	S0, 01
S1	S1, 10	S1, 10	S1, 10	S0, 01
	S*, Q QN			

between adjacent columns of the table. This layout helps our analysis because we assume that only one input changes at a time, and that the circuit always reaches a stable total state before another input changes.

At any time, the circuit is in a particular internal state and a particular input is applied to it; we called this combination the total state of the circuit. Let us start at the stable total state “S0/00” (S = S0, C D = 00), as shown in Figure 7-76. Now suppose that we change D to 1. The total state moves to one cell to the right; we have a new stable total state, S0/01. The D input is different, but the internal state and output are the same as before. Next, let us change C to 1. The total state moves one cell to the right to S0/11, which is unstable. The next-state entry in this cell sends the circuit to internal state S1, so the total state moves down one cell, to S1/11. Examining the next-state entry in the new cell, we find that we have reached a stable total state. We can trace the behavior of the circuit for any desired sequence of single input changes in this way.

Now we can revisit the question of simultaneous input changes. Even though “almost simultaneous” input changes may occur in practice, we must assume that nothing happens simultaneously in order to analyze the behavior of sequential circuits. The impossibility of simultaneous events is supported by the varying delays of circuit components themselves, which depend on voltage, temperature, and fabrication parameters. What this tells us is that a set of n inputs that appear to us to change “simultaneously” may actually change in any of n! different orders from the point of view of the circuit operation.

For example, consider the operation of the D latch as shown in Figure 7-77. Let us assume that it starts in stable total state S1/11. Now suppose that C and D

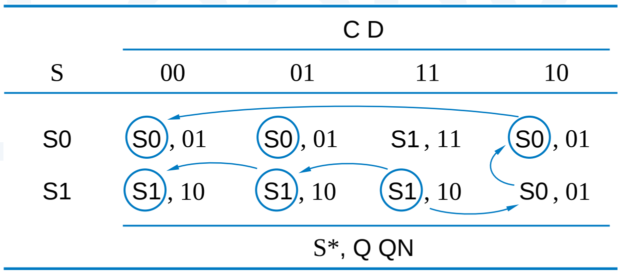


Figure 7-77
Multiple input changes with the D latch.

S	CD = 00	01	11	10
S0	S0, 01	S0, 01	S1, 11	S0, 01
S1	S1, 10	S1, 10	S1, 10	S0, 01
	S*, Q QN			

are both “simultaneously” set to 0. In reality, the circuit behaves as if one or the other input went to 0 first. Suppose that C changes first. Then the sequence of two left-pointing arrows in the table tells us that the circuit goes to stable total state S1/00. However, if D changes first, then the other sequence of arrows tells us that the circuit goes to stable total state S0/00. So the final state of the circuit is unpredictable, a clue that the feedback loop may actually become metastable if we set C and D to 0 simultaneously. The time span over which this view of simultaneity is relevant is the setup- and hold-time window of the D latch.

Simultaneous input changes don’t always cause unpredictable behavior. However, we must analyze the effects of all possible orderings of signal changes to determine this; if all orderings give the same result, then the circuit output is predictable. For example, consider the behavior of the D latch starting in total state S0/00 with C and D simultaneously changing from 0 to 1; it always ends up in total state S1/11.

*7.9.2 Analyzing Circuits with Multiple Feedback Loops

In circuits with multiple feedback loops, we must break all of the loops, creating one fictional buffer and state variable for each loop that we break. There are many possible ways, which mathematicians call *cut sets*, to break the loops in a given circuit, so how do we know which one is best? The answer is that any *minimal cut set*—a cut set with a minimum number of cuts—is fine. Mathematicians can give you an algorithm for finding a minimal cut set, but as a digital designer working on small circuits, you can just eyeball the circuit to find one.

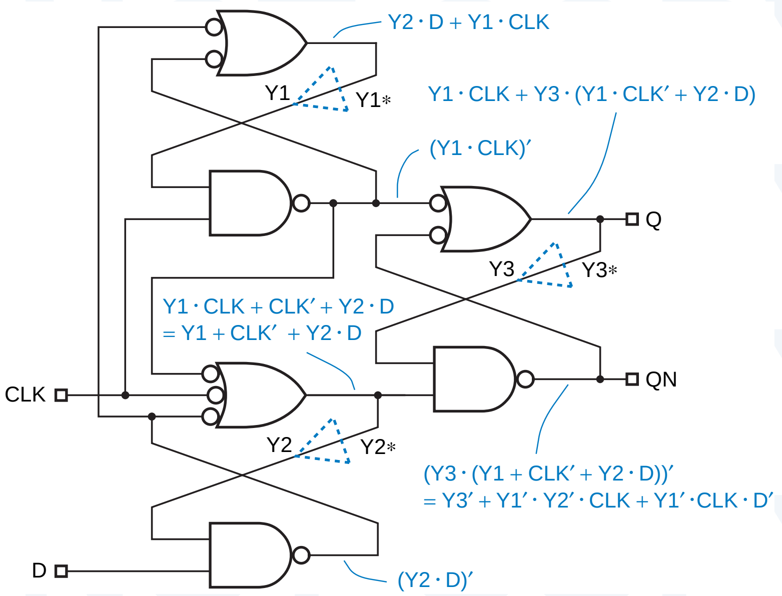
cut set

minimal cut set

Different cut sets for a circuit lead to different excitation equations, transition tables, and state/output tables. However, the stable total states derived from one minimal cut set correspond one-to-one to the stable total states derived from any other minimal cut set for the same circuit. That is, state/output tables derived from different minimal cut sets display the same input/output behavior, with only the names and coding of the states changed.

If you use more than the minimal number of cuts to analyze a feedback sequential circuit, the resulting state/output table will still describe the circuit correctly. However, it will use 2^m times as many states as necessary, where m is the number of extra cuts. Formal state-minimization procedures can be used to reduce this larger table to the proper size, but it’s a much better idea to select a minimal cut set in the first place.

A good example of a sequential circuit with multiple feedback loops is the commercial circuit design for a positive edge-triggered TTL D flip-flop that we showed in Figure 7-20. The circuit is redrawn in simplified form in Figure 7-78, assuming that the original circuit’s PR_L and CLR_L inputs are never asserted, and also showing fictional buffers to break the three feedback loops. These three loops give rise to eight states, compared with the minimum of four states used by the two-loop design in Figure 7-19. We’ll address this curious difference later.



Y2 · D + Y1 · CLK
Y1
Y1*
Y1 · CLK + Y3 · (Y1 · CLK′ + Y2 · D)
(Y1 · CLK)′
Q
Y3
Y3*
Y1 · CLK + CLK′ + Y2 · D
= Y1 + CLK′ + Y2 · D
CLK
QN
Y2
Y2*
(Y3 · (Y1 + CLK′ + Y2 · D))′
= Y3′ + Y1′ · Y2′ · CLK + Y1′ · CLK · D′
D
(Y2 · D)′

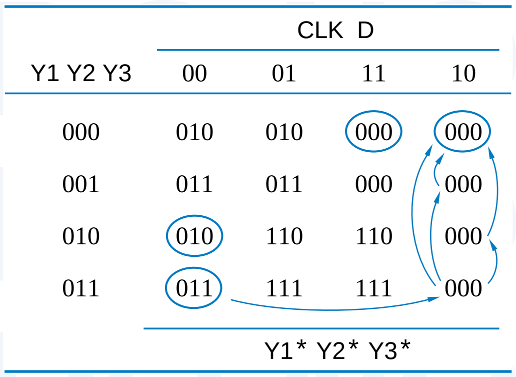
Figure 7-78
Simplified positive edge-triggered D flip-flop for analysis.

The following excitation and output equations can be derived from the logic diagram in Figure 7-78:

$$
\begin{aligned}
Y1* &= Y2 \cdot D + Y1 \cdot CLK \\
Y2* &= Y1 + CLK' + Y2 \cdot D \\
Y3* &= Y1 \cdot CLK + Y1 \cdot Y3 + Y3 \cdot CLK' + Y2 \cdot Y3 \cdot D \\
Q &= Y1 \cdot CLK + Y1 \cdot Y3 + Y3 \cdot CLK' + Y2 \cdot Y3 \cdot D \\
QN &= Y3' + Y1' \cdot Y2' \cdot CLK + Y1' \cdot CLK \cdot D'
\end{aligned}
$$

Figure 7-79
Transition table for the D flip-flop in Figure 7-78.

	CLK D			
Y1 Y2 Y3	00	01	11	10
000	010	010	000	000
001	011	011	000	000
010	010	110	110	000
011	011	111	111	000
100	010	010	111	111
101	011	011	111	111
110	010	110	111	111
111	011	111	111	111
	Y1* Y2* Y3*			



	CLK D			
Y1 Y2 Y3	00	01	11	10
000	010	010	000	000
001	011	011	000	000
010	010	110	110	000
011	011	111	111	000
	Y1* Y2* Y3*			

Figure 7-80 Portion of the D flip-flop transition table showing a noncritical race.

The corresponding transition table is shown in Figure 7-79, with the stable total states circled. Before going further, we must introduce the concept of "races."

*7.9.3 Races

race

In a feedback sequential circuit, a *race* is said to occur when multiple internal variables change state as a result of a single input changing state. In the example of Figure 7-79, a race occurs in stable total state 011/00 when CLK is changed from 0 to 1. The table indicates that the next internal state is 000, a 2-variable change from 011.

As we've discussed previously, logic signals never really change "simultaneously." Thus, the internal state may make the change 011→000 as either 011→001→000 or 011→010→000. Figure 7-80 indicates that the example circuit, starting in total state 011/00, should go to total state 000/10 when CLK changes from 0 to 1. However, it may temporarily visit total state 001/10 or 010/10 along the way. That's OK, because the next internal state for both of these temporary states is 000; therefore, even in the temporary states, the excitation logic continues to drive the feedback loops toward the same stable total state, 000/10. Since the final state does not depend on the order in which the state variables change, this is called a *noncritical race*.

noncritical race

Now suppose that the next-state entry for total state 010/10 is changed to 110, as shown in Table 7-81, and consider the case that we just analyzed. Starting in stable total state 011/00 and changing CLK to 1, the circuit may end up in internal state 000 or 111 depending on the order and speed of the internal variable changes. This is called a *critical race*.

critical race

WATCH OUT FOR CRITICAL RACES! When you design a feedback-based sequential circuit, you must ensure that its transition table does not contain any critical races. Otherwise, the circuit may operate unpredictably, with the next state for racy transitions depending on factors like temperature, voltage, and the phase of the moon.

	CLK D			
Y1 Y2 Y3	00	01	11	10
000	010	010	000	000
001	011	011	000	000
010	010	110	110	110
011	011	111	111	000
100	010	010	111	111
101	011	011	111	111
110	010	110	111	111
111	011	111	111	111
	Y1* Y2* Y3*			

Figure 7-81
A transition table containing a critical race.

*7.9.4 State Tables and Flow Tables

Analysis of the real transition table, Figure 7-79, for our example D flip-flip circuit, shows that it does not have any critical races; in fact, it has no races except the noncritical one we identified earlier. Once we've determined this fact, we no longer need to refer to state variables. Instead, we can name the state-variable combinations and determine the output values for each state/input combination to obtain a state/output table such as Figure 7-82.

Figure 7-82
State/output table for the D flip-flop in Figure 7-78.

	CLK D			
S	00	01	11	10
S0	S2 , 01	S2 , 01	S0 , 01	S0 , 01
S1	S3 , 10	S3 , 10	S0 , 10	S0 , 10
S2	S2 , 01	S6 , 01	S6 , 01	S0 , 01
S3	S3 , 10	S7 , 10	S7 , 10	S0 , 01
S4	S2 , 01	S2 , 01	S7 , 11	S7 , 11
S5	S3 , 10	S3 , 10	S7 , 10	S7 , 10
S6	S2 , 01	S6 , 01	S7 , 11	S7 , 11
S7	S3 , 10	S7 , 10	S7 , 10	S7 , 10
	S* , Q QN			

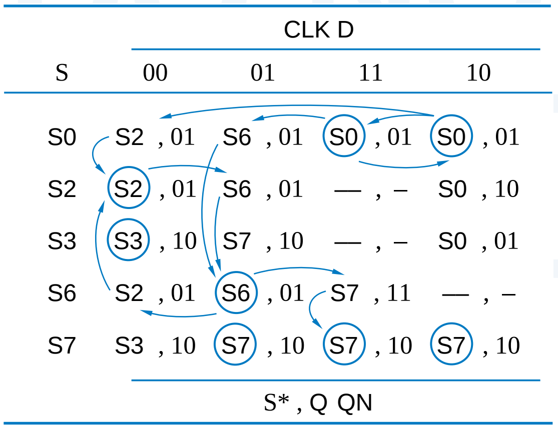
	CLK D			
S	00	01	11	10
S0	S2 , 01	S6 , 01	$\textcircled{S0}$, 01	$\textcircled{S0}$, 01
S2	$\textcircled{S2}$, 01	S6 , 01	— , –	S0 , 10
S3	$\textcircled{S3}$, 10	S7 , 10	— , –	S0 , 01
S6	S2 , 01	$\textcircled{S6}$, 01	S7 , 11	— , –
S7	S3 , 10	$\textcircled{S7}$, 10	$\textcircled{S7}$, 10	$\textcircled{S7}$, 10
	S* , Q QN			

Figure 7-83 Flow and output table for the D flip-flop in Figure 7-78.

The state table shows that for some single input changes, the circuit takes multiple “hops” to get to a new stable total state. For example, in state S0/11, an input change to 01 sends the circuit first to state S2 and then to stable total state S6/01. A *flow table* eliminates multiple hops and shows only the ultimate destination for each transition. The flow table also eliminates the rows for unused internal states—ones that are stable for no input combination—and eliminates the next-state entries for total states that cannot be reached from a stable total state as the result of a single input change. Using these rules, Figure 7-83 shows the flow table for our D flip-flop example.

flow table

The flip-flop’s edge-triggered behavior can be observed in the series of state transitions shown in Figure 7-84. Let us assume that the flip-flop starts in internal state S0/10. That is, the flip-flop is storing a 0 (since Q = 0), CLK is 1, and D is 0. Now suppose that D changes to 1; the flow table shows that we move one cell to the left, still a stable total state with the same output value. We can change D between 0 and 1 as much as we want, and just bounce back and forth between these two cells. However, once we change CLK to 0, we move to



	CLK D			
S	00	01	11	10
S0	S2 , 01	S6 , 01	$\textcircled{S0}$, 01	$\textcircled{S0}$, 01
S2	$\textcircled{S2}$, 01	S6 , 01	— , –	S0 , 10
S3	$\textcircled{S3}$, 10	S7 , 10	— , –	S0 , 01
S6	S2 , 01	$\textcircled{S6}$, 01	S7 , 11	— , –
S7	S3 , 10	$\textcircled{S7}$, 10	$\textcircled{S7}$, 10	$\textcircled{S7}$, 10
	S* , Q QN			

Figure 7-84 Flow and output table showing the D flip-flop’s edge-triggered behavior.

	CLK D			
S	00	01	11	10
SB	SB , 01	S6 , 01	SB , 01	SB , 01
S3	S3 , 10	S7 , 10	— , –	SB , 01
S6	SB , 01	S6 , 01	S7 , 11	— , –
S7	S3 , 10	S7 , 10	S7 , 10	S7 , 10
	S* , Q QN			

Figure 7-85 Reduced flow and output table for a positive edge-triggered D flip-flop.

internal state S2 or S6, depending on whether D was 0 or 1 at the time; but still the output is unchanged. Once again, we can change D between 0 and 1 as much as we want, this time bouncing between S2 and S6 without changing the output.

The moment of truth finally comes when CLK changes to 1. Depending on whether we are in S2 or S6, we go back to S0 (leaving Q at 0) or to S7 (setting Q to 1). Similar behavior involving S3 and S7 can be observed on a rising clock edge that causes Q to change from 1 to 0.

In Figure 7-19, we showed a circuit for a positive edge-triggered D flip-flop with only two feedback loops and hence four states. The circuit that we just analyzed has three loops and eight states. Even after eliminating unused states, the flow table has five states. However, a formal state-minimization procedure can be used to show that states S0 and S2 are "compatible," so that they can be merged into a single state SB that handles the transitions for both original states, as shown in Figure 7-85. Thus, the job really could have been done by a four-state circuit. In fact, in Exercise 7.62 you'll show that the circuit in Figure 7-19 does the job specified by the reduced flow table.

*7.9.5 CMOS D Flip-Flop Analysis

CMOS flip-flops typically use transmission gates in their feedback loops. For example, Figure 7-86 shows the circuit design of the "FD1" positive-edge-triggered D flip-flop in LSI Logic's LCA10000 series of CMOS gate arrays. Such a flip-flop can be analyzed in the same way as a purely logic-gate based design, once you recognize the feedback loops. Figure 7-86 has two feedback loops, each of which has a pair of transmission gates in a mux-like configuration controlled by CLK and CLK′, yielding the following loop equations:

$$\begin{aligned} Y1* &= CLK' \cdot D' + CLK \cdot Y1 \\ Y2* &= CLK \cdot Y1' + CLK' \cdot Y2 \end{aligned}$$

Except for the double inversion of the data as it goes from D to Y2* (once in the Y1* equation and again in the Y2* equation), these equations are very reminiscent of the master/slave-latch structure of the D flip-flop in Figure 7-15.

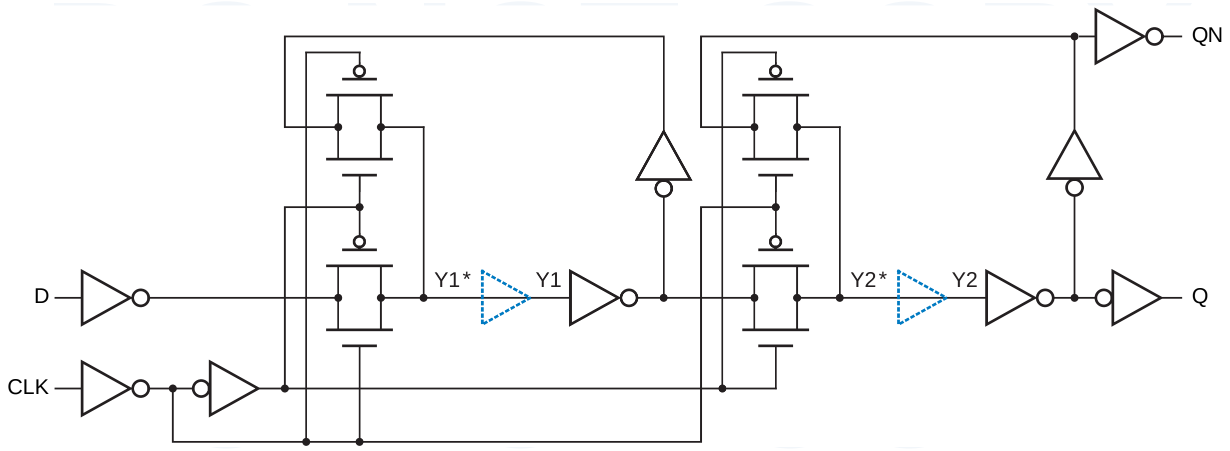


Figure 7-86 Positive edge-triggered CMOS D flip-flop for analysis.

Completing the formal analysis of the circuit is left as an exercise (7.69). Note, however, that since there are just two feedback loops, the resulting state and flow tables will have the minimum of just four states.

FEEDBACK CIRCUIT DESIGN

The feedback sequential circuits that we've analyzed in this section exhibit quite reasonable behavior since, after all, they are latch and flip-flop circuits that have been used for years. However, if we throw together a "random" collection of gates and feedback loops, we won't necessarily get "reasonable" sequential circuit behavior. In a few rare cases, we may not get a sequential circuit at all (see Exercise 7.63), and in many cases, the circuit may be unstable for some or all input combinations (see Exercise 7.68). Thus, the design of feedback sequential circuits continues to be something of a black art, and is practiced only by a small fraction of digital designers. Still, the next section introduces basic concepts that help you do simple designs.

*7.10 Feedback Sequential Circuit Design

It's sometimes useful to design a small feedback sequential circuit, such as a specialized latch or a pulse catcher; this section will show you how. It's even possible that you might go on to be an IC designer, and be responsible for designing high-performance latches and flip-flops from scratch. This section will serve as an introduction to the basic concepts you'll need, but you'll still need considerably more study, experience, and finesse to do it right.

*7.10.1 Latches

Although the design of feedback sequential circuits is generally a hard problem, some circuits can be designed pretty easily. Any circuit with one feedback loop

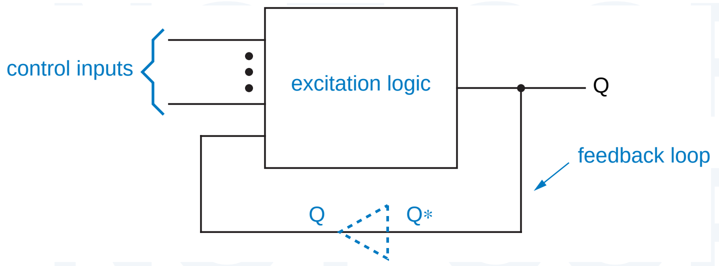


Figure 7-87
General structure of a latch.

is just a variation of an S-R or D latch. It has the general structure shown in Figure 7-87, and an excitation equation with the following format:

$$Q* = \text{(forcing term)} + \text{(holding term)} \cdot Q$$

For example, the excitation equations for S-R and D latches are

$$Q* = S + R' \cdot Q$$
$$Q* = C \cdot D + C' \cdot Q$$

Corresponding circuits are shown in Figure 7-88(a) and (b).

hazard-free excitation logic

In general, *the excitation logic in a feedback sequential circuit must be hazard free;* we'll demonstrate this fact by way of an example. Figure 7-89(a) is a Karnaugh map for the D-latch excitation circuit of Figure 7-88(b). The map exhibits a static-1 hazard when D and Q are 1 and C is changing. Unfortunately, the latch's feedback loop may not hold its value if a hazard-induced glitch occurs. For example, consider what happens when D and Q are 1 and C changes from 1 to 0; the circuit should latch a 1. However, unless the inverter is very fast, the output of the top AND gate goes to 0 before the output of the bottom one goes to 1, the OR-gate output goes to 0, and the feedback loop stores a 0.

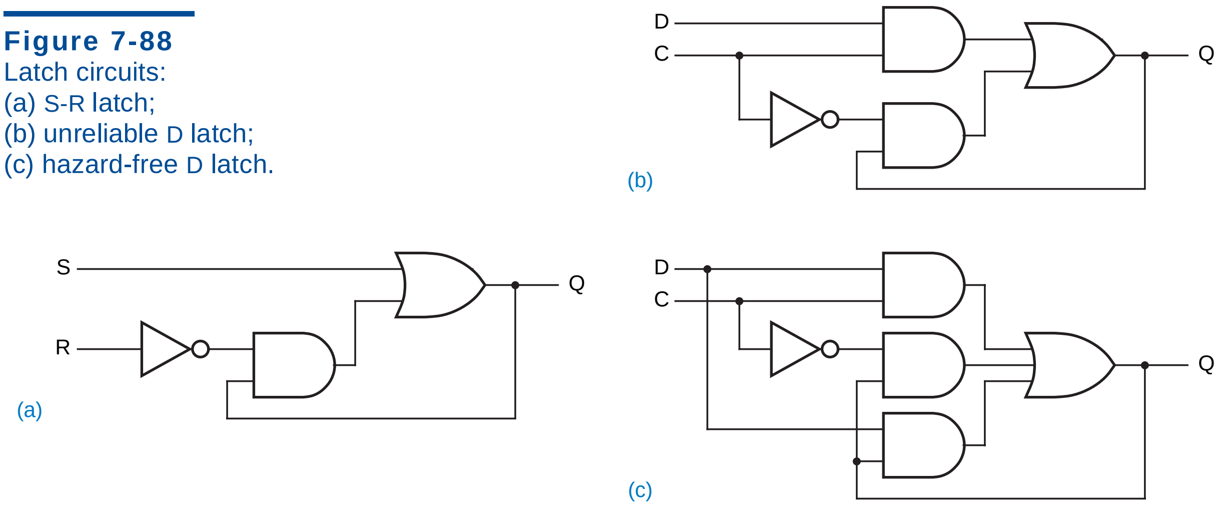


Figure 7-88
Latch circuits:
(a) S-R latch;
(b) unreliable D latch;
(c) hazard-free D latch.

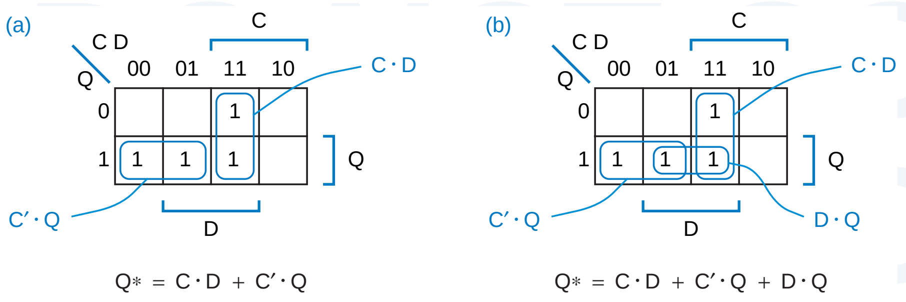


Figure 7-89
Karnaugh maps for D-latch excitation functions: (a) original, containing a static-1 hazard; (b) hazard eliminated.

Hazards can be eliminated using the methods described in Section 4.5. In the D latch, we simply include the consensus term in the excitation equation:

$$Q* \;=\; C \cdot D + C' \cdot Q + D \cdot Q$$

Figure 7-88(c) shows the corresponding hazard-free, correct D-latch circuit.

Now, suppose we need a specialized "D" latch with three data inputs, D1–D3, that stores a 1 only if D1–D3 = 010. We can convert this word description into an excitation equation that mimics the equation for a simple D latch:

$$Q* \;=\; C \cdot (D1' \cdot D2 \cdot D3') + C' \cdot Q$$

Eliminating hazards, we get

$$Q* \;=\; C \cdot D1' \cdot D2 \cdot D3' + C' \cdot Q + D1' \cdot D2 \cdot D3' \cdot Q$$

The hazard-free excitation equation can be realized with discrete gates or in a PLD, as we'll show in Section 8.2.6.

*7.10.2 Designing Fundamental-Mode Flow Table

To design feedback sequential circuits more complex than latches, we must first convert the word description into a flow table. Once we have a flow table, we can turn the crank (with some effort) to obtain a circuit.

When we construct the flow table for a feedback sequential circuit, we give each state a meaning in the context of the problem, much like we did in the

PRODUCT-TERM EXPLOSION

In some cases, the need to cover hazards can cause an explosion in the number of product terms in a two-level realization of the excitation logic. For example, suppose we need a specialized latch with two control inputs, C1 and C2, and three data inputs as before. The latch is to be "open" only if both control inputs are 1, and is to store a 1 if any data input is 1. The minimal excitation equation is

$$\begin{aligned} Q* &= C1 \cdot C2 \cdot (D1 + D2 + D3) + (C1 \cdot C2)' \cdot Q \\ &= C1 \cdot C2 \cdot D1 + C1 \cdot C2 \cdot D2 + C1 \cdot C2 \cdot D3 + C1' \cdot Q + C2' \cdot Q \end{aligned}$$

However, it takes six consensus terms to eliminate hazards (see Exercise 7.71).

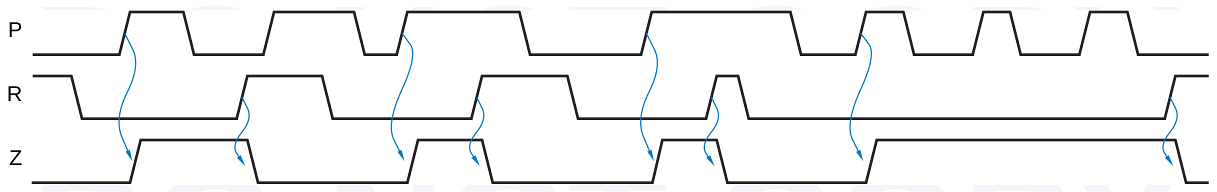


Figure 7-90 Typical functional behavior of a pulse-catching circuit.

design of clocked state machines. However, it's easier to get confused when constructing the flow table for a feedback sequential circuit, because not every total state is stable. Therefore, the recommended procedure is to construct a *primitive flow table*—one that has only one stable total state in each row. Since there is only one stable state per row, the output may be shown as a function of state only.

primitive flow table

In a primitive flow table, each state has a more precise "meaning" than it might otherwise have, and the table's structure clearly displays the underlying fundamental-mode assumption: inputs change one at a time, with enough time between changes for the circuit to settle into a new stable state. A primitive flow table usually has extra states, but we can "turn the crank" later to minimize the number of states once we have a flow table that we believe to be correct.

We'll use the following problem, a "pulse-catching" circuit, to demonstrate flow-table design:

> Design a feedback sequential circuit with two inputs, P (pulse) and R (reset), and a single output Z that is normally 0. The output should be set to 1 whenever a 0-to-1 transition occurs on P, and should be reset to 0 whenever R is 1. Typical functional behavior is shown in Figure 7-90.

Figure 7-91 is a primitive flow table for the pulse catcher. Let's walk through how this table was developed.

We assume that the pulse catcher is initially idle, with P and R both 0; this is the IDLE state, with Z = 0. In this state, if reset occurs (R = 1), we could probably stay in the same state, but since this is supposed to be a *primitive* flow table, we create a new state, RES1, so as not to have two stable total states in the same row. On the other hand, if a pulse occurs (P = 1) in the IDLE state, we definitely want to go to a different state, which we've named PLS1, since we've caught a pulse and we must set the output to 1. Input combination 11 is not considered in the IDLE state, because of the fundamental-mode assumption that only one input changes at a time; we assume the circuit always makes it to another stable state before input combination 11 can occur.

Next, we fill in the next-state entries for the newly created RES1 state. If reset goes away, we can go back to the IDLE state. If a pulse occurs, we must remain in a "reset" state since, according to the timing diagram, a 0-to-1 transition that occurs while R is 1 is ignored. Again, to keep the flow table in primitive form, we must create a new state for this case, RES2.

		P R				
Meaning	S	00	01	11	10	Z
Idle, waiting for pulse	IDLE	IDLE	RES1	—	PLS1	0
Reset, no pulse	RES1	IDLE	RES1	RES2	—	0
Got pulse, output on	PLS1	PLS2	—	RES2	PLS1	1
Reset, got pulse	RES2	—	RES1	RES2	PLSN	0
Pulse gone, output on	PLS2	PLS2	RES1	—	PLS1	1
Got pulse, but output off	PLSN	IDLE	—	RES2	PLSN	0
		S*				

Figure 7-91 Primitive flow table for pulse-catching circuit.

Now that we have one stable total state in each column, we may be able to go to existing states for more transitions, instead of always defining new states. Sure enough, starting in stable total state PLS1/10, for R = 1 we can go to RES2, which fits the requirement of producing a 0 output. On the other hand, where should we go for P = 0? IDLE is a stable total state in the 00 column, but it produces the wrong output value. In PLS1, we've gotten a pulse and haven't seen a reset, so if the pulse goes away, we should go to a state that still has Z = 1. Thus, we must create a new state PLS2 for this case.

In RES2, we can safely go to RES1 if the pulse goes away. However, we've got to be careful if reset goes away, as shown in the timing diagram. Since we've already passed the pulse's 0-to-1 transition, we can't go to the PLS1 state, since that would give us a 1 output. Instead, we create a new state PLSN with a 0 output.

Finally, we can complete the next-state entries for PLS2 and PLSN without creating any new states. Notice that starting in PLS2, we bounce back and forth between PLS2 and PLS1 and maintain a continuous 1 output if we get a series of pulses without an intervening reset input.

*7.10.3 Flow-Table Minimization

As we mentioned earlier, a primitive flow table usually has more states than required. However, there exists a formal procedure, discussed in the References, for minimizing the number of states in a flow table. This procedure is often complicated by the existence of don't-care entries in the flow table.

Fortunately, our example flow table is small and simple enough to minimize by inspection. States IDLE and RES1 produce the same output, and they have the same next-state entry for input combinations where they are both specified. Therefore, they are compatible and may be replaced by a single state

Figure 7-92 Reduced flow table for pulse-catching circuit.

	P R				
S	00	01	11	10	Z
IDLE	IDLE	IDLE	RES	PLS	0
PLS	PLS	IDLE	RES	PLS	1
RES	IDLE	IDLE	RES	RES	0
	S*				

(IDLE) in a reduced flow table. The same can be said for states PLS1 and PLS2 (replaced by PLS) and for RES2 and PLSN (replaced by RES). The resulting reduced flow table, which has only three states, is shown in Figure 7-92.

*7.10.4 Race-Free State Assignment

The next somewhat creative (read "difficult") step in feedback sequential circuit design is to find a race-free assignment of coded states to the named states in the reduced flow table. Recall from Section 7.9.3 that a race occurs when multiple internal variables change state as a result of a single input change. A feedback-based sequential circuit must not contain any critical races; otherwise, the circuit may operate unpredictably. As we'll see, eliminating races often necessitates *increasing* the number of states in the circuit.

state adjacency diagram

A circuit's potential for having races in its transition table can be analyzed by means of a *state adjacency diagram* for its flow table. The adjacency diagram is a simplified state diagram that omits self-loops and does not show the direction of other transitions (A→B is drawn the same as B→A), or the input combinations that cause them. Figure 7-93 is an example fundamental-mode flow table and Figure 7-94(a) is the corresponding adjacency diagram.

adjacent states

Two states are said to be *adjacent* if there is an arc between them in the state adjacency diagram. For race-free transitions, adjacent coded states must differ in only one bit. If two states A and B are adjacent, it doesn't matter whether

Figure 7-93 Example flow table for the state-assignment problem.

	X Y			
S	00	01	11	10
A	A	B	A	B
B	B	B	D	B
C	C	A	A	C
D	D	B	D	C
	S*			

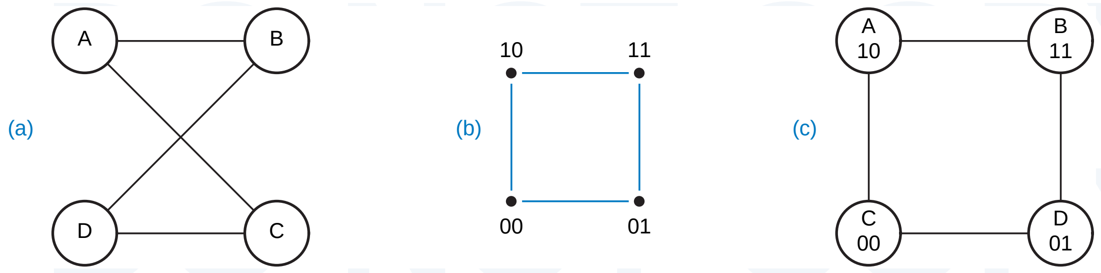


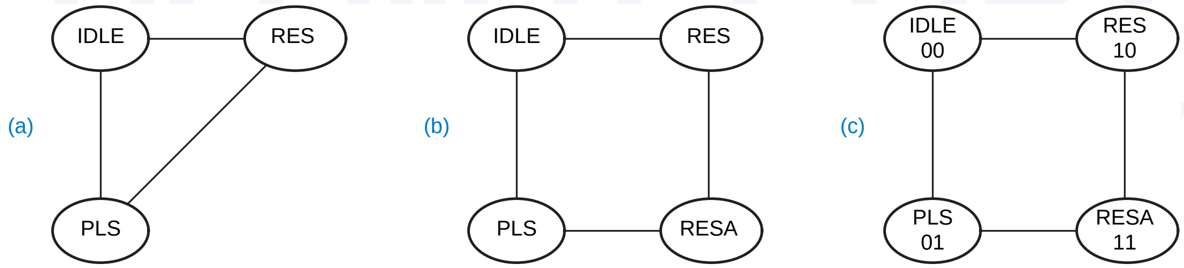
Figure 7-94 State-assignment example: (a) adjacency diagram; (b) a 2-cube; (c) one of eight possible race-free state assignments.

the original flow table had transitions from A to B, from B to A, or both. Any one of these transitions is a race if A and B differ in more than one variable. That's why we don't need to show the direction of transitions in an adjacency diagram.

The problem of finding a race-free assignment of states to *n* state variables is equivalent to the problem of mapping the nodes and arcs of the adjacency diagram onto the nodes and arcs of an *n*-cube. In Figure 7-94, the problem is to map the adjacency diagram (a) onto a 2-cube (b). You can visually identify eight ways to do this (four rotations times two flips), one of which produces the state assignment shown in (c).

Figure 7-95(a) is an adjacency diagram for our pulse-catching circuit, based on the reduced flow table in Figure 7-92. Clearly, there's no way to map this "triangle" of states onto a 2-cube. At this point, we can only go back and modify the original flow table. In particular, the flow table tells us the destination state that we *eventually* must reach for each transition, but it doesn't prevent us from going through other states on the way. As shown in Figure 7-96, we can create a new state RESA and make the transition from PLS to RES by going through RESA. The modified state table has the new adjacency diagram shown in Figure 7-95(b), which has many race-free assignments possible. A transition table based on the assignment in (c) is shown in Figure 7-98. Note that the PLS$\rightarrow$RESA$\rightarrow$RES transition will be slower than the other transitions in the

Figure 7-95 Adjacency diagrams for the pulse catcher: (a) using original flow table; (b) after adding a state; (c) showing one of eight possible race-free state assignments.



	P R				
S	00	01	11	10	Z
IDLE	IDLE	IDLE	RES	PLS	0
PLS	PLS	IDLE	RESA	PLS	1
RESA	—	—	RES	—	–
RES	IDLE	IDLE	RES	RES	0
	S*				

Figure 7-96 State table allowing a race-free assignment for the pulse-catching circuit.

original flow table because it requires two internal state changes, with two propagation delays through the feedback loops.

Even though we added a state in the previous example, we still got by with just two state variables. However, we may sometimes have to add one or more state variables to make a race-free assignment. Figure 7-97(a) shows the worst possible adjacency diagram for four states—every state is adjacent to every other state. Clearly, this adjacency diagram cannot be mapped onto a 2-cube. However, there is a race-free assignment of states to a 3-cube, shown in (b), where each state in the original flow table is represented by two equivalent states in the final state table. Both states in a pair, such as A1 and A2, are equivalent and produce the same output. Each state is adjacent to one of the states in every other pair, so a race-free transition may be selected for each next-state entry.

HANDLING THE GENERAL ASSIGNMENT CASE

In the general case of a flow table with 2^n rows, it can be shown that a race-free assignment can be obtained using $2n$–1 state variables (see References). However, there aren't many applications for fundamental-mode circuits with more than a few states, so the general case is of little more than academic interest.

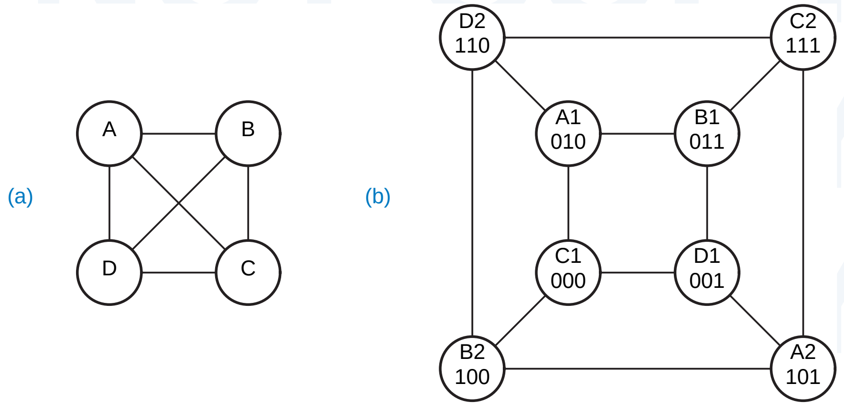


Figure 7-97 A worst-case scenario: (a) 4-state adjacency diagram; (b) assignment using pairs of equivalent states.

	P R				
Y1 Y2	00	01	11	10	Z
00	00	00	10	01	0
01	01	00	11	01	1
11	—	—	10	—	–
10	00	00	10	10	0
	Y1* Y2*				

Figure 7-98 Race-free transition table for the pulse-catching circuit.

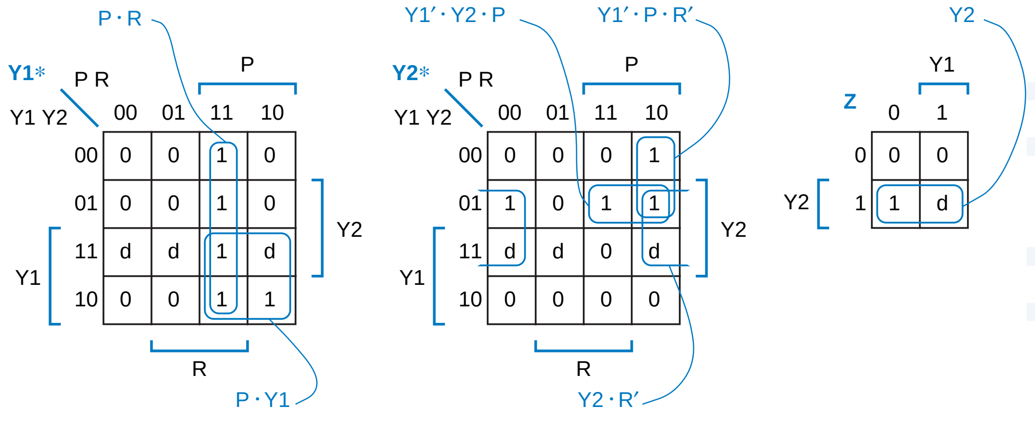
*7.10.5 Excitation Equations

Once we have a race-free transition table for a circuit, we can just "turn the crank" to obtain excitation equations for the feedback loops. Figure 7-99 shows Karnaugh maps derived from Figure 7-98, the pulse catcher's transition table. Notice that "don't-care" next-state and output entries give rise to corresponding entries in the maps, simplifying the excitation and output logic. The resulting minimal sum-of-products excitation and output equations are as follows:

$$
\begin{aligned}
Y1* &= P \cdot R + P \cdot Y1 \\
Y2* &= Y2 \cdot R' + Y1' \cdot Y2 \cdot P + Y1' \cdot P \cdot R' \\
Z &= Y2
\end{aligned}
$$

Recall that the excitation logic in a feedback sequential circuit must be hazard free. The sum-of-products expressions we derived happen to be hazard free as well as minimal. The logic diagram of Figure 7-100 uses these expressions to build the pulse-catching circuit.

Figure 7-99 Karnaugh maps for pulse-catcher excitation and output logic.



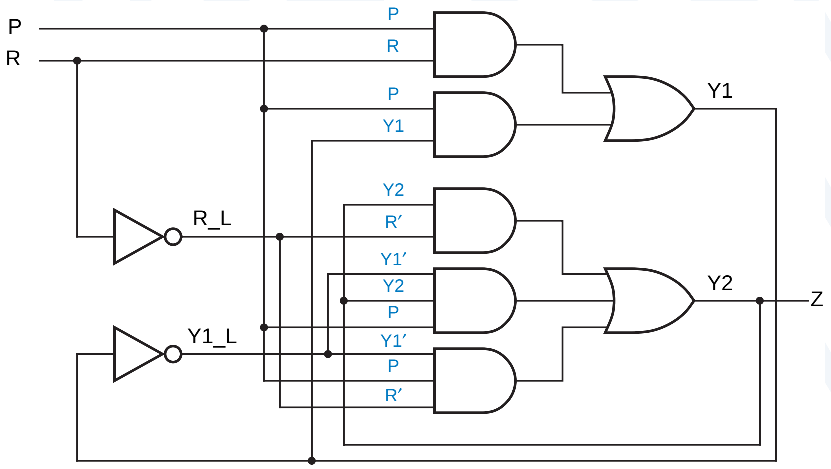


Figure 7-100 Pulse-catching circuit.

*7.10.6 Essential Hazards

After all this effort, you'd think that we'd have a pulse-catching circuit that would operate reliably all of the time. Unfortunately, we're not quite there yet. A fundamental-mode circuit must generally satisfy five requirements for proper operation:

1. Only one input signal may change at a time, with a minimum bound between successive input changes.
2. There must be a maximum propagation delay through the excitation logic and feedback paths; this maximum must be less than the time between successive input changes.
3. The state assignment (transition table) must be free of critical races.
4. The excitation logic must be hazard free.
5. The minimum propagation delay through the excitation logic and feedback paths must be greater than the maximum timing skew through the "input logic."

Without the first requirement, it would be impossible to satisfy the major premise of fundamental-mode operation—that the circuit has time to settle into a stable total state between successive input changes. The second requirement says that the excitation logic is fast enough to do just that. The third requirement ensures that the proper state changes are made even if the excitation circuits for different state variables have different delays. The fourth requirement ensures that state variables that aren't supposed to change on a particular transition don't.

The last requirement deals with subtle timing-dependent errors that can occur in fundamental-mode circuits, even ones that satisfy the first four requirements. An *essential hazard* is the possibility of a circuit going to an

essential hazard

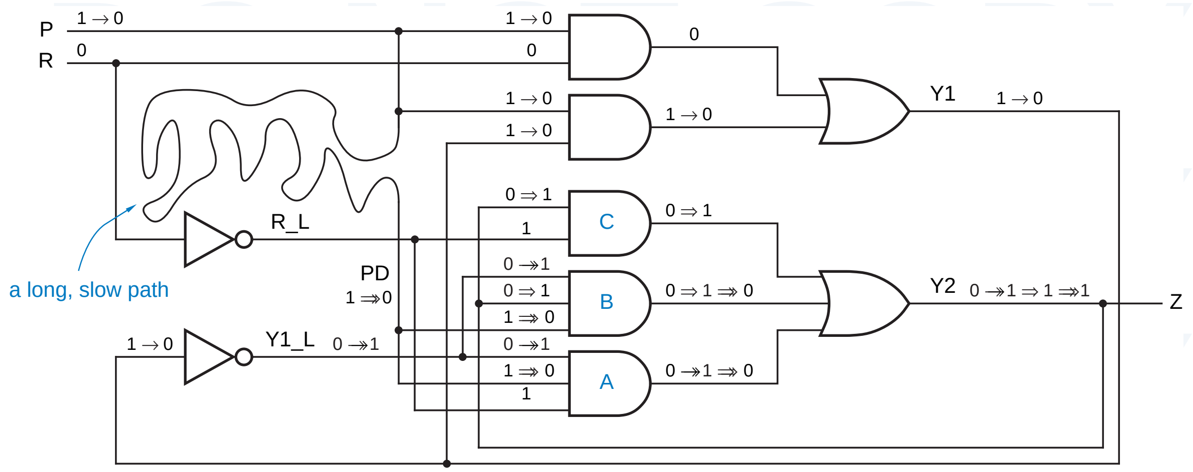


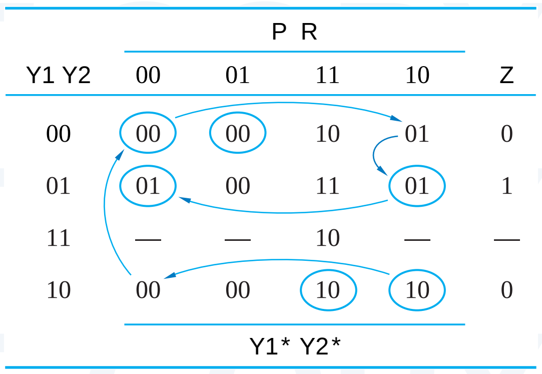
Figure 7-101 Physical conditions in pulse-catching circuit for exhibiting an essential hazard.

erroneous next state as the result of a single input change; the error occurs if the input change is not seen by all of the excitation circuits before the resulting state-variable transition(s) propagate back to the inputs of the excitation circuits. In a world where "faster is better" is the usual rule, a designer may sometimes have to *slow down* excitation logic to mask these hazards.

Essential hazards are best explained in terms of an example, our pulse-catching circuit. Suppose we built our circuit on a PCB or a chip, and we (or, more likely, our CAD system) inadvertently connected input signal P through a long, slow path at the point shown in Figure 7-101. Let's assume that this delay is longer than the propagation delay of the AND-OR excitation logic.

Now consider what can happen if P R = 10, the circuit is in internal state 10, and P changes from 1 to 0. According to the transition table, repeated in Figure 7-102, the circuit should go to internal state 00, and that's that. But let's look at the actual operation of the circuit, as traced in Figure 7-101:

- (Changes shown with "⟶") The first thing that happens after P changes is that Y1 changes from 1 to 0. Now the circuit is in internal state 00.
- (Changes shown with "↠") Y1_L changes from 0 to 1. The change in Y1_L at AND gate A causes its output to go to 1, which in turn forces Y2 to 1. Whoops, now the circuit is in internal state 01.
- (Changes shown with "⇒") The change in Y2 at AND gates B and C causes their outputs to go to 1, reinforcing the 1 output at Y2. All this time, we've been waiting for the 1-to-0 change in P to appear at point PD.
- (Changes shown with "⇛") Finally, PD changes from 1 to 0, forcing the outputs of AND gates A and B to 0. However, AND gate C still has a 1 output, and the circuit remains in state 01—*the wrong state*.



	P R				
Y1 Y2	00	01	11	10	Z
00	00	00	10	01	0
01	01	00	11	01	1
11	—	—	10	—	—
10	00	00	10	10	0
	Y1* Y2*				

Figure 7-102 Transition table for the pulse-catching circuit, exhibiting an essential hazard.

The only way to avoid this erroneous behavior in general is to ensure that changes in P arrive at the inputs of all the excitation circuits before any changes in state variables do. Thus, the inevitable difference in input arrival times, called *timing skew*, must be less than the propagation delay of the excitation circuits and feedback loops. This timing requirement can generally be satisfied only by careful design *at the electrical circuit level*.

timing skew

In the example circuit, it would appear that the hazard is easily masked, even by non-electrical engineers, since the designer need only ensure that a straight wire has shorter propagation delay than an AND-OR structure, easy in most technologies.

Still, many feedback sequential circuits, such as the TTL edge-triggered D flip-flop in Figure 7-19, have essential hazards in which the input skew paths include inverters. In such cases, the input inverters must be guaranteed to be faster than the excitation logic; that's not so trivial in either board-level or IC design. For example, if the excitation circuit in Figure 7-101 were physically built using AND-OR-INVERT gates, the delay from input changes to Y1_L could be very short indeed, as short as the delay through a single inverter.

Essential hazards can be found in most but not all fundamental-mode circuits. There's an easy rule for detecting them; in fact, this is the definition of "essential hazard" in some texts:.

- A fundamental-mode flow table contains an essential hazard for a stable total state S and an input variable X if, starting in state S, the stable total state reached after three successive transitions in X is different from the stable total state reached after one transition in X.

THESE HAZARDS ARE, WELL, ESSENTIAL!

Essential hazards are called "essential" because they are inherent in the flow table for a particular sequential function, and will appear in any circuit realization of that function. They can be masked only by controlling the delays in the circuit. Compare with static hazards in combinational logic, where we could eliminate hazards by adding consensus terms to a logic expression.

Thus, the essential hazard in the pulse catcher is detected by the arrows in Figure 7-102, starting in internal state 10 with P R = 10.

A fundamental-mode circuit must have at least three states to have an essential hazard, so latches don't have them. On the other hand, all flip-flops (circuits that sample inputs on a clock edge) do.

*7.10.7 Summary

In summary, you use the following steps to design a feedback sequential circuit:

1. Construct a primitive flow table from the circuit's word description.
2. Minimize the number of states in the flow table.
3. Find a race-free assignment of coded states to named states, adding auxiliary states or splitting states as required.
4. Construct the transition table.
5. Construct excitation maps and find a hazard-free realization of the excitation equations.
6. Check for essential hazards. Modify the circuit if necessary to ensure that minimum excitation and feedback delays are greater than maximum inverter or other input-logic delays.
7. Draw the logic diagram.

Also note that some circuits routinely violate the basic fundamental-mode assumption that inputs change one at a time. For example, in a positive-edge-triggered D flip-flop, the D input may change at the same time that CLK changes from 1 to 0, and the flip-flop still operates properly. The same thing certainly cannot be said at the 0-to-1 transition of CLK. Such situations require analysis of the transition table and circuit on a case-by-case basis if proper operation in "special cases" is to be guaranteed.

A FINAL QUESTION

Given the difficulty of designing fundamental-mode circuits that work properly, let alone ones that are fast or compact, how did anyone ever come up with the 6-gate, 8-state, commercial D flip-flop design in Figure 7-20? Don't ask me, I don't know!

7.11 ABEL Sequential-Circuit Design Features

7.11.1 Registered Outputs

ABEL has several features that support the design of sequential circuits. As we'll show in Section 8.3, most PLD outputs can be configured by the user to be *registered outputs* that provide a D flip-flop following the AND-OR logic, as in Figure 7-103. To configure one or more outputs to be registered, an ABEL

registered output

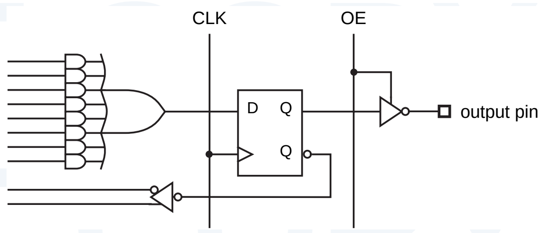


Figure 7-103
PLD registered output.

reg

program's pin declarations normally must contain an `istype` clause using the keyword "`reg`" (rather than "`com`") for each registered output. Table 7-22 is an example program that has three registered outputs and two combinational outputs.

.CLK
.OE
.PR
.RE

As suggested by Figure 7-103, a registered output has at least two other attributes associated with it. The three-state buffer driving the output pin has an output-enable input OE, and the flip-flop itself has a clock input CLK. As shown in Table 7-22, the signals that drive these inputs are specified in the equations section of the program. Each input signal is specified as the corresponding main output signal name followed by an attribute suffix, *.CLK* or *.OE*. Some PLDs have flip-flops with additional controllable inputs; for example, preset and clear inputs have attribute suffixes `.PR` and `.RE` (reset). And some PLDs provide flip-flop types other than D; their inputs are specified with suffixes like `.J` and `.K`.

clocked assignment operator, :=

Within the `equations` section of the ABEL program, the logic values for registered outputs are established using the *clocked assignment operator, :=*. When the PLD is compiled, the expression on the right-hand-side will be applied to the D input of the output flip-flop. All of the same rules as for combinational outputs apply to output polarity control, don't-cares, and so on. In Table 7-22, the state bits Q1–Q3 are registered outputs, so they use clocked assignment, "`:=`". The UNLK and HINT signals are Mealy outputs, combinational functions of current state and input, so they use unclocked assignment, "=". A machine with pipelined outputs (Figure 7-37 on page 454), would instead use clocked assignment for such outputs.

clocked truth-table operator, :>

ABEL's truth-table syntax (Table 4-16 on page 255) can also be used with registered outputs. The only difference is that "`->`" operator between input and output items is changed to "`:>`".

IS `istype` ESSENTIAL?

Older devices, such as the PAL16Rx family, contain a fixed, preassigned mix of combinational and registered outputs and are not configurable. With these devices, the compiler can deduce each output's type from its pin number and the `istype` statement is not necessary. Even with configurable outputs, some compilers can correctly the output type from the equations. Still, it's a good idea to include the `istype` information anyway, both as a double check and to enhance design portability.

```
module CombLock
Title 'Combination-Lock State Machine'

" Input and Outputs
X, CLOCK       pin;
UNLK, HINT     pin istype 'com';
Q1, Q2, Q3     pin istype 'reg';

Q = [Q1..Q3];

Equations

Q.CLK = CLOCK; Q.OE = 1;

" State variables
Q1 := Q1 & !Q2 & X  #  !Q1 & Q2 & Q3 & !X  #  Q1 & Q2 & !Q3;
Q2 := !Q2 & Q3 & X  #  Q2 & !Q3 & X;
Q3 := Q1 & !Q2 & !Q3  #  Q1 & Q3 & !X  #  !Q2 & !X
      #  !Q1 & !Q3 & !X  #  Q2 & !Q3 & X;

" Mealy outputs
UNLK = Q1 & Q2 & Q3 & !X;
HINT = !Q1 & !Q2 & !Q3 & !X  #  Q1 & !Q2 & X  #  !Q2 & Q3 & X
       #  Q2 & Q3 & !X  #  Q2 & !Q3 & X;

end CombLock
```

Table 7-22 ABEL program using registered outputs.

You can also design feedback sequential circuits in ABEL, without using any of the language's sequential-circuit features. For example, in Section 8.2.6 we show how to specify latches using ABEL.

7.11.2 State Diagrams

The state-machine example in the previous subsection is just a transcription of the combination-lock machine that we synthesized by hand in Section 7.4.6 beginning on page 484. However, most PLD programming languages have a notation for defining, documenting, and synthesizing state machines directly, without ever writing a state, transition, or excitation table or deriving excitation equations by hand. Such a notation is called a *state-machine description language*. In ABEL, this notation is called a "state diagram," and the ABEL compiler does all the work of generating excitation equations that realize the specified machine.

state-machine description language

In ABEL, the keyword *state_diagram* indicates the beginning of a state-machine definition. Table 7-23 shows the textual structure of an ABEL "state diagram." Here *state-variables* is an ABEL set that lists the state variables of the machine. If there are *n* variables in the set, then the machine has 2^n possible states corresponding to the 2^n different assignments of constant values to

state_diagram

state-variables

Table 7-23
Structure of a "state diagram" in ABEL.

```
state_diagram state-variables
state state-value 1 : transition statement;
state state-value 2 : transition statement;
...
state state-value 2^n : transition statement;
```

variables in the set. States are usually given symbolic names in an ABEL program; this makes it easy to try different assignments simply by changing the constant definitions.

state
state-value
equation

An equation for each state variable is developed according to the information in the "state diagram." The keyword `state` indicates that the next states and current outputs for a particular current state are about to be defined; a *state-value* is a constant that defines state-variable values for the current state. A *transition statement* defines the possible next states for the current state.

GOTO statement
IF statement

ABEL has two commonly used transition statements. The *GOTO statement* unconditionally specifies the next state, for example "`GOTO INIT`". The *IF statement* defines the possible next states as a function of an arbitrary logic expressions. (There's also a seldom-used `CASE` statement which we don't cover.)

Table 7-24 shows the syntax of the ABEL `IF` statement. Here *TrueState* and *FalseState* are state values that the machine will go to if *LogicExpression* is true or false, respectively. These statements can be nested: *FalseState* can itself be another `IF` statement, and *TrueState* can be an `IF` statement if it is enclosed in braces. When multiple next states are possible, a nested `IF-THEN-ELSE` structure eliminates the ambiguities that can occur in hand-drawn state diagrams, where the transition conditions leaving a state can overlap (Section 7.5).

Our first example using ABEL's "state diagram" capability is based on our first state-machine design example from Section 7.4.1 on page 466. A state table for this machine was developed in Figure 7-49 on page 469. It is adapted to ABEL in Table 7-25. Several characteristics of this program should be noted:

- The definition of `QSTATE` uses three variables to encode state.
- The definitions of `INIT`–`XTRA3` determine the individual state encodings.
- `IF-THEN-ELSE` statements are nested. A particular next state may appear in multiple places in one set of nested `IF-THEN-ELSE` clauses (e.g., see states `OK0` and `OK1`).
- Expressions like "`(B==1)*(A==0)`" were used instead of equivalents like "`B*!A`" only because the former are a bit more readable.

Table 7-24
Structure of an ABEL `IF` statement.

```
IF LogicExpression THEN
   TrueState;
ELSE
   FalseState;
```

Table 7-25
An example of ABEL's state-diagram notation.

```
module SMEX1
title 'PLD Version of Example State Machine'

" Input and output pins
CLOCK, RESET_L, A, B            pin;
Q1..Q3                          pin istype 'reg';
Z                               pin istype 'com';

" Definitions
QSTATE = [Q1,Q2,Q3];           " State variables
INIT   = [ 0, 0, 0];
A0     = [ 0, 0, 1];
A1     = [ 0, 1, 0];
OK0    = [ 0, 1, 1];
OK1    = [ 1, 0, 0];
XTRA1  = [ 1, 0, 1];
XTRA2  = [ 1, 1, 0];
XTRA3  = [ 1, 1, 1];
RESET = !RESET_L;

state_diagram QSTATE

state INIT: IF RESET THEN INIT
            ELSE IF (A==0) THEN A0
            ELSE A1;

state A0:   IF RESET THEN INIT
            ELSE IF (A==0) THEN OK0
            ELSE A1;

state A1:   IF RESET THEN INIT
            ELSE IF (A==0) THEN A0
            ELSE OK1;

state OK0:  IF RESET THEN INIT
            ELSE IF (B==1)&(A==0) THEN OK0
            ELSE IF (B==1)&(A==1) THEN OK1
            ELSE IF (A==0) THEN OK0
            ELSE IF (A==1) THEN A1;

state OK1:  IF RESET THEN INIT
            ELSE IF (B==1)&(A==0) THEN OK0
            ELSE IF (B==1)&(A==1) THEN OK1
            ELSE IF (A==0) THEN A0
            ELSE IF (A==1) THEN OK1;

state XTRA1: GOTO INIT;
state XTRA2: GOTO INIT;
state XTRA3: GOTO INIT;

equations

QSTATE.CLK = CLOCK; QSTATE.OE = 1;
Z = (QSTATE == OK0) # (QSTATE == OK1);

END SMEX1
```

Table 7-26
Reduced equations for SMEX1 PLD.

```
Q1 := (!Q2.FB & !Q3.FB & RESET_L
     # Q1.FB & RESET_L);
Q1.C = (CLOCK);
Q1.OE = (1);

Q2 := (Q1.FB & !Q3.FB & RESET_L & !A
     # Q1.FB & Q3.FB & RESET_L & A
     # Q1.FB & Q2.FB & RESET_L & B);
Q2.C = (CLOCK);
Q2.OE = (1);

Q3 := (!Q2.FB & !Q3.FB & RESET_L & A
     # Q1.FB & RESET_L & A);
Q3.C = (CLOCK);
Q3.OE = (1);

Z = (Q2 & Q1);
```

- The first IF statement in each of states INIT–OK1 ensures that the machine goes to the INIT state if RESET is asserted.
- Next-state equations are given for XTRA1–XTRA3 to ensure that the machine goes to a "safe" state if it somehow gets into an unused state.
- The single equation in the "equations" section of the program determines the behavior of the Moore-type output.

Table 7-26 shows the resulting excitation and output equations produced by ABEL compiler (the reverse-polarity equations are not shown). Notice the use of variable names like "Q1.FB" in the right-hand sides of the equations. Here, the ".FB" attribute suffix refers to the "feedback" signal into the AND-OR array coming from the flip-flop's Q output. This is done to make it clear that the signal is coming from the flip-flop, not from the corresponding PLD output pin, which can be selected in some complex PLDs. As shown in Figure 7-104, ABEL actually allows you to select among three possible values on the right-hand side of an equation using an attribute suffix on the signal name:

USE IT OR ELSE

ABEL's IF-THEN-ELSE structure eliminates the transition ambiguity that can occur in state diagrams. However, the ELSE clause of an IF statement is optional. If it is omitted, the next state for some input combinations will be unspecified. Usually this is not the designer's intention.

Nevertheless, if you can guarantee that the unspecified input combinations will never occur, you may be able to reduce the size of the transition logic. If the @DCSET directive is given, the ABEL compiler treats the transition outputs for the unspecified state/input combinations as "don't-cares." In addition, it treats *all* transitions out of unused states as "don't-cares."

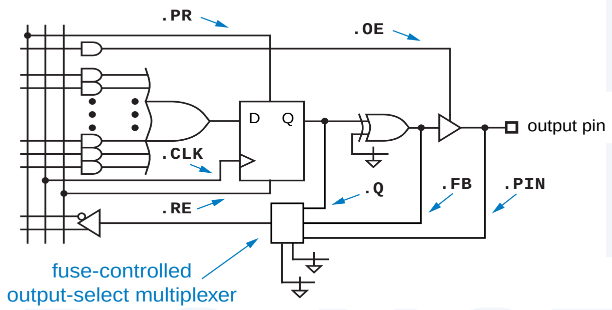


Figure 7-104 Output selection capability in a complex PLD.

`.Q`	The actual flip-flop output pin before any programmable inversion.
`.FB`	A value equal to the value that the output pin would have if enabled.
`.PIN`	The actual signal at the PLD output pin. This signal is floating or driven by another device if the three-state driver is not enabled.

.Q

.FB

.PIN

Obviously, the `.PIN` value should not be used in a state-machine excitation equation since it is not guaranteed always to equal the state variable.

Despite the use of "high-level language," the program's author still had to refer to the original, hand-constructed state table in Figure 7-49 to come up with ABEL version in Table 7-25. A different approach is shown in Table 7-27. This program was developed directly from the word description of the state machine, which is repeated below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> – A had the same value at each of the two previous clock ticks, *or*
>
> – B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

A key idea in the new approach is to remove the last value of `A` from the state definitions, and instead to have a separate flip-flop that keeps track of it (`LASTA`). Then only two non-`INIT` states must be defined: `LOOKING` ("still looking for a match") and `OK` ("got a match or `B` has been 1 since last match"). The `Z` output is a simple combinational decode of the `OK` state.

PHANTOM (OF THE) OPERAND

Real CPLDs typically have only a two-input output-select multiplexer and omit the `.FB` input shown in Figure 7-104. When an equation calls for a signal with the `.FB` attribute, the ABEL compiler uses the corresponding `.Q` signal and simply adjusts it with the appropriate inversion (or not).

Table 7-27
A more “natural” ABEL program for the example state machine.

```
module SMEX2
title 'Alternate Version of Example State Machine'

" Input and output pins
CLOCK, RESET_L, A, B            pin;
LASTA, Q1, Q2                   pin istype 'reg';
Z                               pin istype 'com';

" Definitions
QSTATE  = [Q1,Q2];               " State variables
INIT    = [ 0, 0];               " State encodings
LOOKING = [ 0, 1];
OK      = [ 1, 0];
XTRA    = [ 1, 1];
RESET = !RESET_L;

state_diagram QSTATE

state INIT:    IF RESET THEN INIT ELSE LOOKING;

state LOOKING: IF RESET THEN INIT
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state OK:      IF RESET THEN INIT
               ELSE IF B THEN OK
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state XTRA:    GOTO INIT;

equations
LASTA.CLK = CLOCK; QSTATE.CLK = CLOCK; QSTATE.OE = 1;

LASTA := A;
Z = (QSTATE == OK);

END SMEX2
```

*7.11.3 External State Memory

In some situations, the state memory of a PLD-based state machine may be kept in flip-flops external to the PLD. ABEL provides a special version of the `state_diagram` statement to handle this situation:

`state_diagram` *current-state-variables* -> *next-state variables*

current-state-variables
next-state-variables

Here *current-state-variables* is an ABEL set that lists the input signals which represent the current state of the machine, and *next-state-variables* is a set that lists the corresponding output signals which are the excitation for external D flip-flops holding the state of the machine, for example,

```
state_diagram [CURQ1, CURQ2] -> [NEXTQ1, NEXTQ2]
state_diagram state-variables
state state-value 1 :
        optional equation;
        optional equation;
        . . .
        transition statement;
state state-value 2 :
        optional equation;
        optional equation;
        . . .
        transition statement;
. . .
state state-value 2^n :
        optional equation;
        optional equation;
        . . .
        transition statement;
```

Table 7-28 Structure of an ABEL state diagram with Moore outputs defined.

*7.11.4 Specifying Moore Outputs

The output Z in our example state machine is a Moore output, a function of state only, and we defined this output in Tables 7-25 and 7-27 using an appropriate equation in the equations section of the program. Alternatively, ABEL allows Moore outputs to be specified along with the state definitions themselves. The transition statement in a state definition may be preceded by one or more optional equations, as shown in Table 7-28. To use this capability with the machine in Table 7-27, for example, we would eliminate the Z equation in the equations section, and rewrite the state diagram as shown in Table 7-29.

As in other ABEL equations, when a variable such as Z appears on the left-hand side of multiple equations, the right-hand sides are OR'ed together to form the final result (as discussed in Section 4.6.3). Also notice that Z is still specified

```
state_diagram QSTATE

state INIT:    Z = 0;
               IF RESET THEN INIT ELSE LOOKING;

state LOOKING: Z = 0;
               IF RESET THEN INIT
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state OK:      Z = 1;
               IF RESET THEN INIT
               ELSE IF B THEN OK
               ELSE IF (A == LASTA) THEN OK
               ELSE LOOKING;

state XTRA:    Z = 0;
               GOTO INIT;
```

Table 7-29 State machine with embedded Moore output definitions.

as a combinational, not registered, output. If `Z` were a registered output, the desired output value would occur one clock tick after the machine visited the corresponding state.

*7.11.5 Specifying Mealy and Pipelined Outputs with `WITH`

Some state-machine outputs are functions of the inputs as well as state. In Section 7.3.2, we called them Mealy outputs or pipelined outputs, depending on whether they occurred immediately upon an input change or only after a clock edge. ABEL's *`WITH` statement* provides a way to specify these outputs side-by-side with the next states, rather than separately in the `equations` section of the program.

`WITH` statement

As shown in Table 7-30, the syntax of the `WITH` statement is very simple. Any next-state value which is part of a transition statement can be followed by the keyword `WITH` and a bracketed list of equations that are "executed" for the specified transition. Formally, let "`E`" an excitation expression that is true only when the specified transition is to be taken. Then for each equation in the `WITH`'s bracketed list, the right-hand side is AND'ed with `E` and assigned to the left-hand side. The equations can use either unclocked or clocked assignment to create Mealy or pipelined outputs, respectively.

Table 7-30 Structure of ABEL `WITH` statement.

```
next-state WITH {
       equation;
       equation;
       ...
}
```

We developed an example "combination lock" state machine with Mealy outputs in Table 7-14 on page 484. The same state machine is specified by the ABEL program in Table 7-31, using `WITH` statements for the Mealy outputs. Note that closing brackets take the place of the semicolons that normally end the transition statements for the states.

Based on the combination lock's word description, it is not possible to realize `UNLK` and `HINT` as pipelined outputs, since they depend on the current value of `X`. However, if we redefine `UNLK` to be asserted for the entire "unlocked" state, and `HINT` to be the actual recommended next value of `X`, we can create a new machine with pipelined outputs, as shown in Table 7-32. Notice that we used the clocked assignment operator for the outputs. More importantly, notice that the values of `UNLK` and `HINT` are different than in the Mealy example, since they have to "look ahead" one clock tick.

Because of "lookahead," pipelined outputs can be more difficult than Mealy outputs to design and understand. In the example above, we even had to modify the problem statement to accommodate them. The advantage of

Table 7-31
State machine with embedded Mealy output definitions.

```
module SMEX4
title 'Combination-Lock State Machine'

" Input and output pins
CLOCK, X                        pin;
Q1..Q3                          pin istype 'reg';
UNLK, HINT                      pin istype 'com';

" Definitions
S        = [Q1,Q2,Q3];          " State variables
ZIP      = [ 0, 0, 0];          " State encodings
X0       = [ 0, 0, 1];
X01      = [ 0, 1, 0];
X011     = [ 0, 1, 1];
X0110    = [ 1, 0, 0];
X01101   = [ 1, 0, 1];
X011011  = [ 1, 1, 0];
X0110111 = [ 1, 1, 1];

state_diagram S

state ZIP:      IF X==0 THEN X0    WITH {UNLK = 0; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

state X0:       IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X01           WITH {UNLK = 0; HINT = 1}

state X01:      IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X011          WITH {UNLK = 0; HINT = 1}

state X011:     IF X==0 THEN X0110 WITH {UNLK = 0; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

state X0110:    IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X01101        WITH {UNLK = 0; HINT = 1}

state X01101:   IF X==0 THEN X0    WITH {UNLK = 0; HINT = 0}
                ELSE X011011       WITH {UNLK = 0; HINT = 1}

state X011011:  IF X==0 THEN X0110 WITH {UNLK = 0; HINT = 0}
                ELSE X0110111      WITH {UNLK = 0; HINT = 1}

state X0110111: IF X==0 THEN X0    WITH {UNLK = 1; HINT = 1}
                ELSE ZIP           WITH {UNLK = 0; HINT = 0}

equations
S.CLK = CLOCK;

END SMEX4
```

pipelined outputs is that, since they are connected directly to register outputs, they are valid a few gate-delays sooner after a state change than Moore or Mealy outputs, which normally include additional combinational logic. In the combination-lock example, it's probably not that important to open your lock or see your hint a few nanoseconds earlier. However, shaving off a few gate delays can be quite important in high-speed applications.

Table 7-32 State machine with embedded pipelined output definitions.

```
module SMEX5
title 'Combination-Lock State Machine'

" Input and output pins
CLOCK, X                        pin;
Q1..Q3                          pin istype 'reg';
UNLK, HINT                      pin istype 'reg';

" Definitions
S        = [Q1,Q2,Q3];          " State variables
ZIP      = [ 0, 0, 0];          " State encodings
X0       = [ 0, 0, 1];
X01      = [ 0, 1, 0];
X011     = [ 0, 1, 1];
X0110    = [ 1, 0, 0];
X01101   = [ 1, 0, 1];
X011011  = [ 1, 1, 0];
X0110111 = [ 1, 1, 1];

state_diagram S

state ZIP:      IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

state X0:       IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X01           WITH {UNLK := 0; HINT := 1}

state X01:      IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X011          WITH {UNLK := 0; HINT := 0}

state X011:     IF X==0 THEN X0110 WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

state X0110:    IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X01101        WITH {UNLK := 0; HINT := 1}

state X01101:   IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE X011011       WITH {UNLK := 0; HINT := 1}

state X011011:  IF X==0 THEN X0110 WITH {UNLK := 0; HINT := 1}
                ELSE X0110111      WITH {UNLK := 1; HINT := 0}

state X0110111: IF X==0 THEN X0    WITH {UNLK := 0; HINT := 1}
                ELSE ZIP           WITH {UNLK := 0; HINT := 0}

equations
S.CLK = CLOCK; UNLK.CLK = CLOCK; HINT.CLK = CLOCK;

END SMEX5
```

7.11.6 Test Vectors

`.C.`, clock edge

Test vectors for sequential circuits in ABEL have the same uses and limitations as test vectors for combinational circuits, as described in Section 4.6.7. One important addition to their syntax is the use of the constant "`.C.`" to denote a clock edge, 0→1→0. Thus, Table 7-33 is an ABEL program, with test vectors, for a simple 8-bit register with a clock-enable input. A variety of vectors are used to test loading and holding different input values.

```
module REG8EN
title '8-bit register with clock enable'

" Input and output pins
CLK, EN, D1..D8        pin;
Q1..Q8                 pin istype 'reg';

" Sets
D = [D1..D8];
Q = [Q1..Q8];

equations

Q.CLK = CLK;

WHEN EN == 1 THEN Q := D ELSE Q := Q;

test_vectors ([CLK, EN,  D  ] -> [ Q  ])
              [.C.,  1, ^h00] -> [^h00]; " 0s in every bit
              [.C.,  0, ^hFF] -> [^h00]; " Hold capability, EN=0
              [.C.,  1, ^hFF] -> [^hFF]; " 1s in every bit
              [.C.,  0, ^h00] -> [^hFF]; " Hold capability
              [.C.,  1, ^h55] -> [^h55]; " Adjacent bits shorted
              [.C.,  0, ^hAA] -> [^h55]; " Hold capability
              [.C.,  1, ^hAA] -> [^hAA]; " Adjacent bits shorted
              [.C.,  1, ^h55] -> [^h55]; " Load with quick setup
              [.C.,  1, ^hAA] -> [^hAA]; " Again
END REG8EN
```

Table 7-33 ABEL program with test vectors for a simple 8-bit register.

A typical approach to testing state machines is to write vectors that not only cause the machine to visit every state, but also to exercise every transition from every state. A key difference and challenge compared to combinational-circuit test vectors is that the vectors must first drive the machine into the desired state before testing a transition, and then come back again for each different transition from that state.

Thus, Table 7-34 shows test vectors for the state machine in Table 7-27. It's important to understand that, unlike combinational vectors, these vectors work only if applied in exactly the order they are written. Notice that the vectors were written to be independent of the state encoding. As a result, they don't have to be modified if the state encoding is changed.

We encounter another challenge if we attempt to create test vectors for the combination-lock state machine of Table 7-31 on page 539. This machine has a major problem when it comes to testing—it has no reset input. Its starting state at power-up may be different in PLD devices and technologies—the individual flip-flops may be all set, all reset, or all in random states. In the machine's actual application, we didn't necessarily need a reset input, but for testing purposes we somehow have to get to a known starting state.

synchronizing sequence

Luckily, the combination-lock machine has a *synchronizing sequence*—a fixed sequence of one or more input values that will always drive it to a certain known state. In particular, starting from any state, if we apply X=1 to the

Table 7-34 Test vectors for the state machine in Table 7-27.

```
test_vectors
([RESET_L, CLOCK, A, B] -> [QSTATE , LASTA, Z])
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check -->INIT (RESET)
 [   0    ,  .C. , 1, 0] -> [INIT    ,   1   , 0]; "  and LASTA flip-flop
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Come out of initialization
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check LOOKING-->INIT (RESET)
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Come out of initialization
 [   1    ,  .C. , 1, 0] -> [LOOKING,   1   , 0]; " --> LOOKING since 0!=1
 [   1    ,  .C. , 1, 0] -> [OK      ,   1   , 1]; " --> OK since 1==1
 [   0    ,  .C. , 0, 0] -> [INIT    ,   0   , 0]; " Check OK-->INIT (RESET)
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " Go back towards OK ...
 [   1    ,  .C. , 0, 0] -> [OK      ,   0   , 1]; " --> OK since 0==0
 [   1    ,  .C. , 1, 1] -> [OK      ,   1   , 1]; " --> OK since B, even though 1!=0
 [   1    ,  .C. , 1, 0] -> [OK      ,   1   , 1]; " --> OK since 1==1
 [   1    ,  .C. , 0, 0] -> [LOOKING,   0   , 0]; " --> LOOKING since 0!=1
```

machine for four ticks, we will always be in state ZIP by the fourth tick. This is the approach taken by the first four vectors in Table 7-35. Until we get to the known state, we indicate the next-state on the right-hand side of the vector as being “don’t care,” so the simulator or the physical device tester will not flag a random state as an error.

Once we get going, we encounter something else that’s new—Mealy outputs that must be tested. As shown by the fourth and fifth vectors, we don’t have to transition the clock in every test vector. Instead, we can keep the clock fixed at 0, where the last transition left it, and observe the Mealy output values produced by the two input values of X. Then we can test the next state transition.

For the state transitions, we list the expected next state but we show the output values as don’t-cares. For a correct test vector, the outputs must show the values attained *after* the transition, a function of the *next* state. Although it’s possible to figure them out and include them, the complexity is enough to give you a headache, and they will be tested by the next CLOCK=0 vectors anyway.

Creating test vectors for a state machine by hand is a painstaking process, and no matter how careful you are, there’s no guarantee that you’ve tested all its functions and potential hardware faults. For example, the vectors in Table 7-34 do not test (A LASTA) = 10 in state LOOKING, or (A B LASTA) = 100 in state OK. Thus, generating a complete set of test vectors for fault-detection purposes is a process best left to an automatic test-generation program. In Table 7-35, we

SYNCHRONIZING SEQUENCES AND RESET INPUTS

We lucked out with the combination-lock; not all state machines have synchronizing sequences. This is why most state machines are designed with a reset input, which in effect allows a synchronizing sequence of length one.

Table 7-35 Test vectors for the combination-lock state machine of Table 7-31.

```
test_vectors
([CLOCK, X] -> [   S    , UNLK, HINT])
 [ .C. , 1] -> [.X.     , .X. , .X. ]; " Since no reset input, apply
 [ .C. , 1] -> [.X.     , .X. , .X. ]; "   a 'synchronizing sequence'
 [ .C. , 1] -> [.X.     , .X. , .X. ]; "   to reach a known starting
 [ .C. , 1] -> [ZIP     , .X. , .X. ]; "   state
 [  0  , 0] -> [ZIP     ,  0  ,  1  ]; " Test Mealy outputs for both
 [  0  , 1] -> [ZIP     ,  0  ,  0  ]; "   values of X
 [ .C. , 1] -> [ZIP     , .X. , .X. ]; " Test ZIP-->ZIP (X==1)
 [ .C. , 0] -> [X0      , .X. , .X. ]; "   and ZIP-->X0 (X==0)
 [  0  , 0] -> [X0      ,  0  ,  0  ]; " Test Mealy outputs for both
 [  0  , 1] -> [X0      ,  0  ,  1  ]; "   values of X
 [ .C. , 0] -> [X0      , .X. , .X. ]; " Test X0-->X0 (X==0)
 [ .C. , 1] -> [X01     , .X. , .X. ]; "   and X0-->X01 (X==1)
 [  0  , 0] -> [X01     ,  0  ,  0  ]; " Test Mealy outputs for both
 [  0  , 1] -> [X01     ,  0  ,  1  ]; "   values of X
 [ .C. , 0] -> [X0      , .X. , .X. ]; " Test X01-->X0 (X==0)
 [ .C. , 1] -> [X01     , .X. , .X. ]; " Get back to X01
 [ .C. , 1] -> [X011    , .X. , .X. ]; " Test X01-->X011 (X==1)
```

petered out after writing vectors for the first few states; completing the test vectors is left as an exercise (7.87). Still, on the functional testing side, writing a few vectors to exercise the machine's most basic functions can weed out obvious design errors early in the process. More subtle design errors are best detected by a thorough system-level simulation.

7.12 VHDL Sequential-Circuit Design Features

References

The problem of metastability has been around for a long time. Greek philosophers wrote about the problem of indecision thousands of years ago. A group of modern philosophers named Devo sang about metastability in the title song of their *Freedom of Choice* album. And the U.S. Congress still can't decide how to "save" Social Security.

Scan capability can also be added to D latches; see McCluskey.

Most ASICs and MSI-based designs use the sequential-circuit types described in this chapter. However, there are other types that are used in both older discrete designs %(going all the way back to vacuum-tube logic) and in modern, custom VLSI designs.

pulse-mode circuit
pulse input

For example, clocked synchronous state machines are a special case of a more general class of *pulse-mode circuits*. Such circuits have one or more *pulse inputs* such that (a) only one pulse occurs at a time; (b) nonpulse inputs are stable when a pulse occurs; (c) only pulses can cause state changes; and (d) a pulse causes at most one state change. In clocked synchronous state machines, the clock is the single pulse input, and a "pulse" is the triggering edge of the clock. However, it is also possible to build circuits with multiple pulse inputs, and it is possible to use storage elements other than the familiar edge-triggered flip-flops. These possibilities are discussed thoroughly by Edward J. McCluskey in *Logic Design Principles* (Prentice Hall, 1986).

two-phase latch machine

A particularly important type of pulse-mode circuit discussed by McCluskey and others is the *two-phase latch machine*. The rationale for a two-phase clocking approach in VLSI circuits is discussed by Carver Mead and Lynn Conway in *Introduction to VLSI Systems* (Addison-Wesley, 1980).

Fundamental-mode circuits need not use feedback loops as the memory elements. For example, McCluskey's *Introduction to the Theory of Switching Circuits* (McGraw-Hill, 1965) gives several examples of fundamental-mode circuits built from SR flip-flops. His 1986 book shows how transmission gates are used to create memory in CMOS latches and flip-flops, and such circuits are analyzed.

Methods for reducing both completely and incompletely-specified state tables are described in advanced logic design texts, including McCluskey's 1986 book. A more mathematical discussion of these methods and other "theoretical" topics in sequential machine design appears in *Switching and Finite Automata Theory*, 2nd ed., by Zvi Kohavi (McGraw-Hill, 1978).

As we showed in this chapter, improperly constructed state diagrams may yield an ambiguous description of next-state behavior. The “IF-THEN-ELSE” structures in HDLs like ABEL and VHDL can eliminate these ambiguities, but they were not the first to do so. *Algorithmic-state-machine (ASM)* notation, a flowchart-like equivalent of nested IF-THEN-ELSE statements, has been around for over 25 years.

algorithmic state machine (ASM)

So-called *ASM charts* were pioneered at Hewlett-Packard Laboratories by Thomas E. Osborne and were further developed by Osborne’s colleague Christopher R. Clare in a book, *Designing Logic Systems Using State Machines* (McGraw-Hill, 1973). Design and synthesis methods using ASM charts subsequently found a home in many digital design texts, including *The Art of Digital Design* by F. P. Prosser and D. E. Winkel (Prentice-Hall, 1987, 2nd ed.) and *Digital Design* by M. Morris Mano (Prentice-Hall, 1984), as well as in the first two editions of this book. Another notation for describing state machines, an extension of “traditional” state-diagram notation, is the mnemonic documented state (MDS) diagram developed by William I. Fletcher in *An Engineering Approach to Digital Design* (Prentice-Hall, 1980). All of these pioneering methods have now been largely replaced by HDLs and their compilers.

ASM chart

Say something about CAD state-diagram entry tools. Too bad they’re not ASM.

Drill Problems

7.1 Give three examples of metastability that occur in everyday life, other than ones discussed in this chapter.

7.2 Sketch the outputs of an SR latch of the type shown in Figure 7-5 for the input waveforms shown in Figure 7.2. Assume that input and output rise and fall times are zero, that the propagation delay of a NOR gate is 10 ns, and that each time division below is 10 ns.

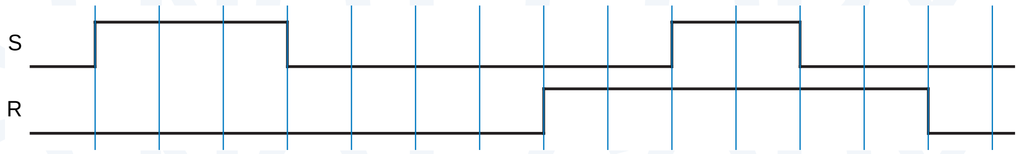


Figure X7.2

7.3 Repeat Drill 7.2 using the input waveforms shown in Figure 7.3. Although you may find the result unbelievable, this behavior can actually occur in real devices whose transition times are short compared to their propagation delay.

7.4 Figure 7-34 showed how to build a T flip-flop with enable using a D flip-flop and combinational logic. Show how to build a D flip-flop using a T flip-flop with enable and combinational logic.

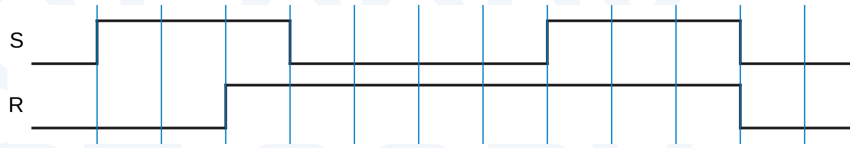


Figure X7.3

7.5 Show how to build a JK flip-flop using a T flip-flop with enable and combinational logic.

7.6 Show how to build an SR latch using a single 74x74 positive-edge-triggered D flip-flop and *no* other components.

7.7 Show how to build a flip-flop equivalent to the 74x109 positive-edge-triggered JKN flip-flop using a 74x74 positive-edge-triggered D flip-flop and one or more gates from a 74x00 package.

7.8 Show how to build a flip-flop equivalent to the 74x74 positive-edge-triggered D flip-flop using a 74x109 positive-edge-triggered JKN flip-flop and *no* other components.

7.9 Analyze the clocked synchronous state machine in Figure 7.9. Write excitation equations, excitation/transition table, and state/output table (use state names A–D for Q1 Q2 = 00–11).

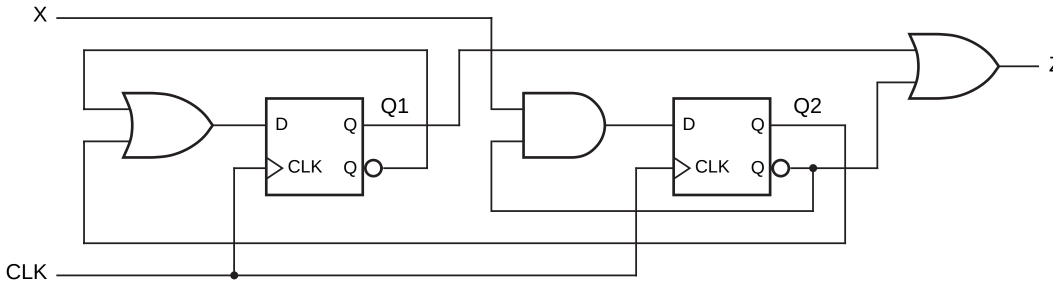


Figure X7.9

7.10 Repeat Drill 7.9, swapping AND and OR gates in the logic diagram. Is the new state/output table the "dual" of the original one? Explain.

7.11 Draw a state diagram for the state machine described by Table 7-6.

7.12 Draw a state diagram for the state machine described by Table 7-12.

7.13 Draw a state diagram for the state machine described by Table 7-14.

7.14 Construct a state and output table equivalent to the state diagram in Figure 7.14. Note that the diagram is drawn with the convention that the state does not change except for input conditions that are explicitly shown.

7.15 Analyze the clocked synchronous state machine in Figure 7.15. Write excitation equations, excitation/transition table, and state table (use state names A–H for Q2 Q1 Q0 = 000–111).

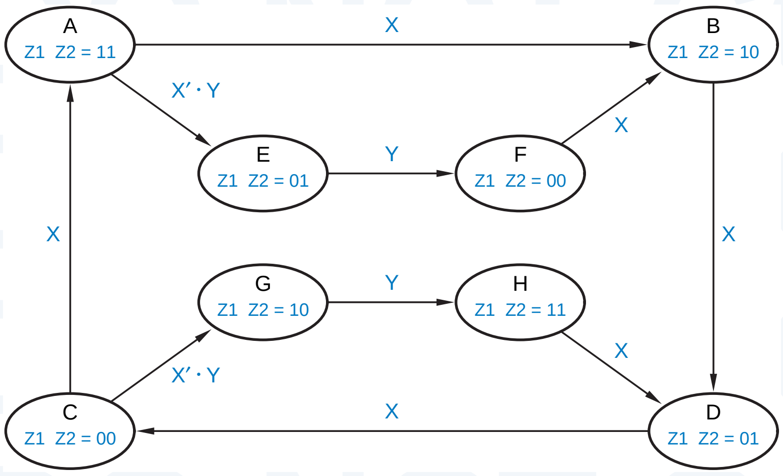


Figure X7.14

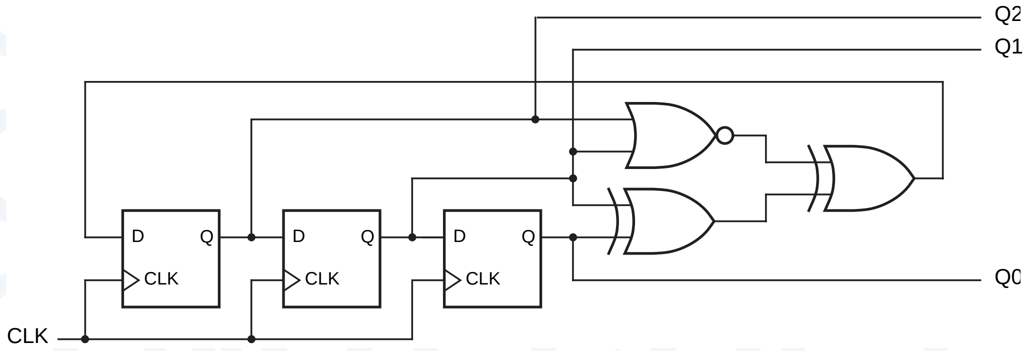


Figure X7.15

7.16 Analyze the clocked synchronous state machine in Figure 7.16. Write excitation equations, excitation/transition table, and state/output table (use state names A–H for Q1 Q2 Q3 = 000–111).

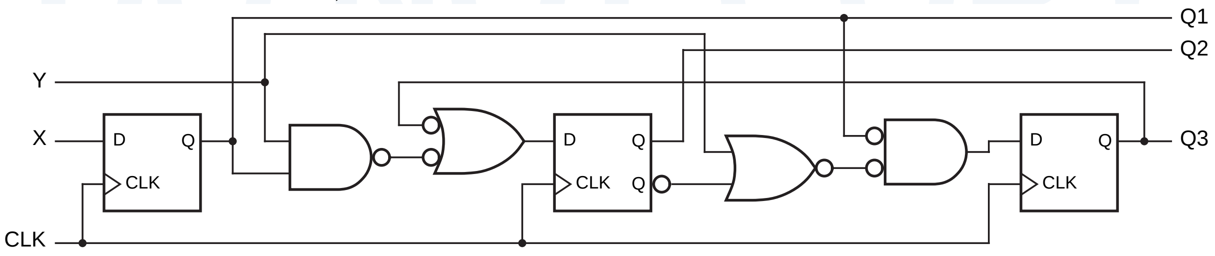


Figure X7.16

7.17 Analyze the clocked synchronous state machine in Figure 7.17. Write excitation equations, transition equations, transition table, and state/output table (use state names A–D for Q1 Q2 = 00–11). Draw a state diagram, and draw a timing diagram for CLK, X, Q1, and Q2 for 10 clock ticks, assuming that the machine starts in state 00 and X is continuously 1.

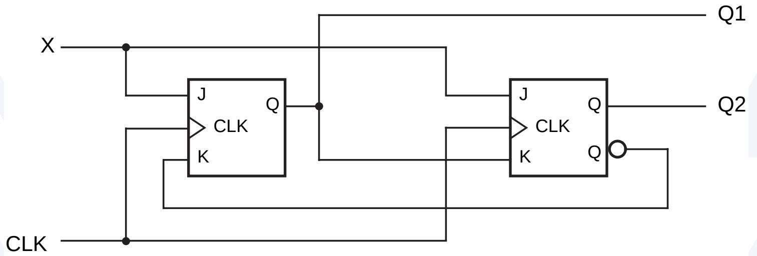


Figure X7.17

7.18 Analyze the clocked synchronous state machine in Figure 7.18. Write excitation equations, transition equations, transition table, and state/output table (use state names A–D for Q1 Q0 = 00–11). Draw a state diagram, and draw a timing diagram for CLK, EN, Q1, and Q0 for 10 clock ticks, assuming that the machine starts in state 00 and EN is continuously 1.

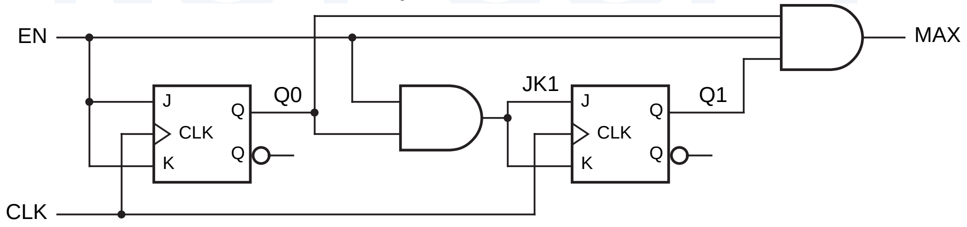


Figure X7.18

7.19 Analyze the clocked synchronous state machine in Figure 7.19. Write excitation equations, excitation/transition table, and state/output table (use state names A–D for Q1 Q2 = 00–11).

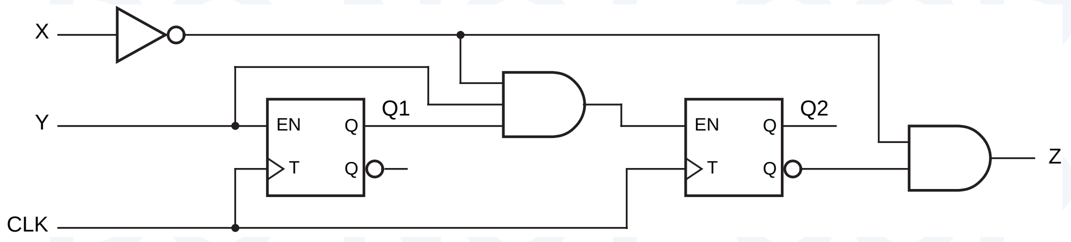


Figure X7.19

7.20 All of the state diagrams in Figure X7.20 are ambiguous. List all of the ambiguities in these state diagrams. (*Hint:* Use Karnaugh maps where necessary to find uncovered and double-covered input combinations.)

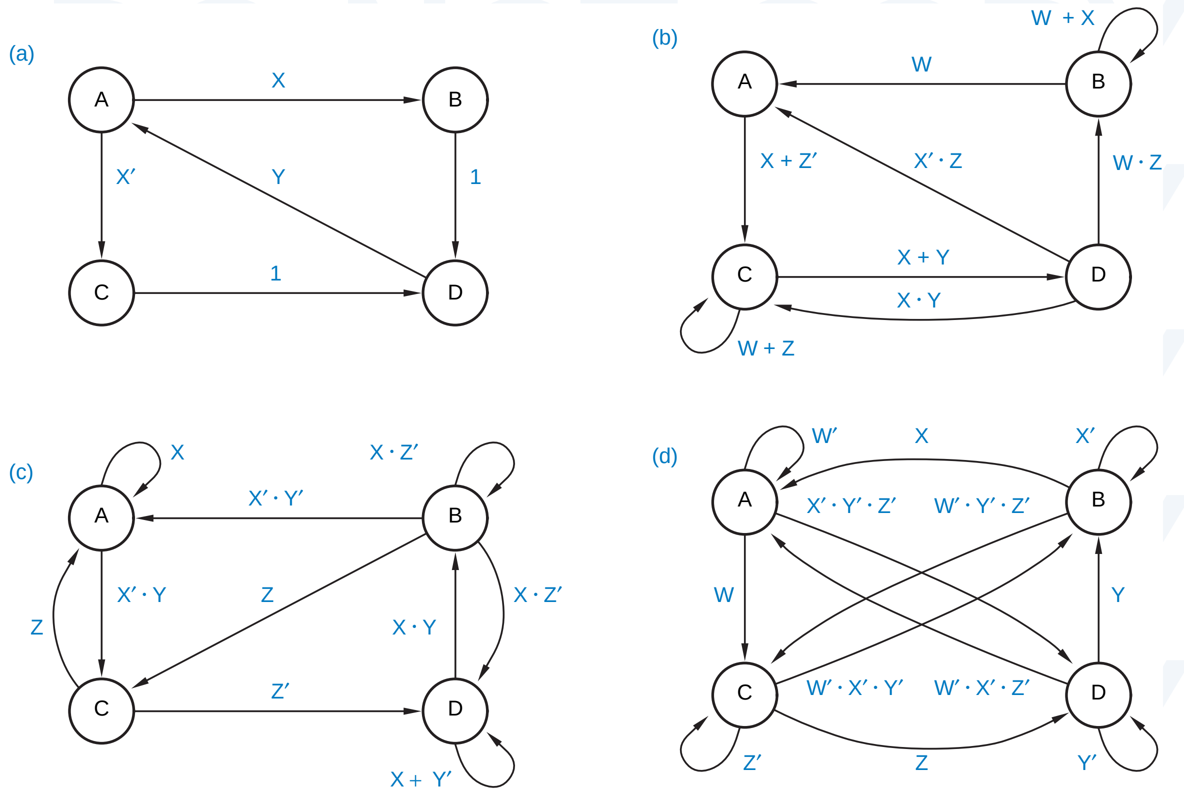


Figure X7.20

7.21 Synthesize a circuit for the state diagram of Figure 7-64 using six variables to encode the state, where the LA–LC and RA–RC outputs equal the state variables themselves. Write a transition list, a transition equation for each state variable as a sum of p-terms, and simplified transition/excitation equations for a realization using D flip-flops. Draw a circuit diagram using SSI and MSI components.

7.22 Starting with the transition list in Table 7-18, find a minimal sum-of-products expression for Q2∗, assuming that the next states for the unused states are true don't-cares.

7.23 Modify the state diagram of Figure 7-64 so that the machine goes into hazard mode immediately if LEFT and RIGHT are asserted simultaneously during a turn. Write the corresponding transition list.

Exercises

7.24 Explain how metastability occurs in a D latch when the setup and hold times are not met, analyzing the behavior of the feedback loop inside the latch.

7.25 What is the minimum setup time of a pulse-triggered flip-flop such as a master/slave J-K or S-R flip-flop? (*Hint:* It depends on certain characteristics of the clock.)

7.26 Describe a situation, other than the metastable state, in which the Q and /Q outputs of a 74x74 edge-triggered D flip-flop may be noncomplementary for an arbitrarily long time.

7.27 Compare the circuit in Figure 7.27 with the D latch in Figure 7-12. Prove that the circuits function identically. In what way is Figure 7.27, which is used in some commercial D latches, better?

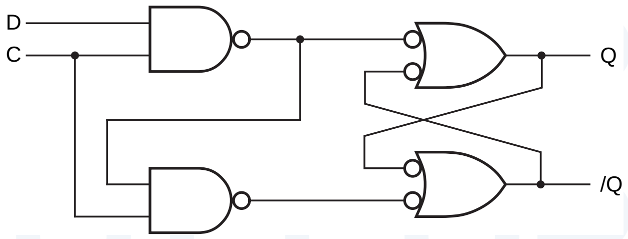


Figure X7.27

7.28 Suppose that a clocked synchronous state machine with the structure of Figure 7-35 is designed using D latches with active-high C inputs as storage elements. For proper next-state operation, what relationships must be satisfied among the following timing parameters?

t_{Fmin}, t_{Fmax}	Minimum and maximum propagation delay of the next-state logic.
t_{CQmin}, t_{CQmax}	Minimum and maximum clock-to-output delay of a D latch.
t_{DQmin}, t_{DQmax}	Minimum and maximum data-to-output delay of a D latch.
t_{setup}, t_{hold}	Setup and hold times of a D latch.
t_{H}, t_{L}	Clock HIGH and LOW times.

7.29 Redesign the state machine in Drill 7.9 using just three inverting gates—NAND or NOR—and no inverters.

7.30 Draw a state diagram for a clocked synchronous state machine with two inputs, INIT and X, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for two successive ticks and 1 for two successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again. Your state diagram should be neatly drawn and planar (no crossed lines). (*Hint:* No more than ten states are required).

7.31 Design a clocked synchronous state machine that checks a serial data line for even parity. The circuit should have two inputs, SYNC and DATA, in addition to CLOCK, and one Moore-type output, ERROR. Devise a state/output table that does the job using just four states, and include a description of each state's meaning in the table. Choose a 2-bit state assignment, write transition and excitation equations, and draw the logic diagram. Your circuit may use D flip-flops, J-K flip-flops, or one of each.

7.32 Design a clocked synchronous state machine with the state/output table shown in Table 7.32, using D flip-flops. Use two state variables, Q1 Q2, with the state assignment A = 00, B = 01, C = 11, D = 10.

Table X7.32

	X		
S	0	1	Z
A	B	D	0
B	C	B	0
C	B	A	1
D	B	C	0
	S*		

7.33 Repeat Exercise 7.32 using J-K flip-flops.

7.34 Write a new transition table and derive minimal-cost excitation and output equations for the state table in Table 7-6 using the “simplest” state assignment in Table 7-7 and D flip-flops. Compare the cost of your excitation and output logic (when realized with a two-level AND-OR circuit) with the circuit in Figure 7-54.

7.35 Repeat Exercise 7.34 using the “almost one-hot” state assignment in Table 7-7.

7.36 Suppose that the state machine in Figure 7-54 is to be built using 74LS74 D flip-flops. What signals should be applied to the flip-flop preset and clear inputs?

7.37 Write new transition and excitation tables and derive minimal-cost excitation and output equations for the state table in Table 7-6 using the “simplest” state assignment in Table 7-7 and J-K flip-flops. Compare the cost of your excitation and output logic (when realized with a two-level AND-OR circuit) with the circuit in Figure 7-56.

7.38 Repeat Exercise 7.37 using the “almost one-hot” state assignment in Table 7-7.

7.39 Construct an application table similar to Table 7-10 for each of the following flip-flop types: (a) S-R; (b) T with enable; (c) D with enable. Discuss the unique problem that you encounter when trying to make the most efficient use of don’t-cares with one of these flip-flops.

7.40 Construct a new excitation table and derive minimal-cost excitation and output equations for the state machine of Table 7-8 using T flip-flops with enable inputs (Figure 7-33). Compare the cost of your excitation and output logic (when realized with a two-level AND-OR circuit) with the circuit in Figure 7-54.

7.41 Determine the full 8-state table of the circuit in Figure 7-54. Use the names U1, U2, and U3 for the unused states (001, 010, and 011). Draw a state diagram and explain the behavior of the unused states.

7.42 Repeat Exercise 7.41 for the circuit of Figure 7-56.

7.43 Write a transition table for the nonminimal state table in Figure 7-51(a) that results from assigning the states in binary counting order, INIT–OKA1 = 000–110. Write corresponding excitation equations for D flip-flops, assuming a minimal-cost disposition of the unused state 111. Compare the cost of your equations with the minimal-cost equations for the minimal state table presented in the text.

7.44 Write the application table for a T flip-flop with enable.

7.45 In many applications, the outputs produced by a state machine during or shortly after reset are irrelevant, as long as the machine begins to behave correctly a short time after the reset signal is removed. If this idea is applied to Table 7-6, the INIT state can be removed and only two state variables are needed to code the remaining four states. Redesign the state machine using this idea. Write a new state table, transition table, excitation table for D flip-flops, minimal-cost excitation and output equations, and logic diagram. Compare the cost of the new circuit with that of Figure 7-54.

7.46 Repeat Exercise 7.45 using J-K flip-flops, and use Figure 7-56 to compare cost.

7.47 Redesign the 1s-counting machine of Table 7-12, assigning the states in binary counting order (S0–S3 = 00, 01, 10, 11). Compare the cost of the resulting sum-of-products excitation equations with the ones derived in the text.

7.48 Repeat Exercise 7.47 using J-K flip-flops.

7.49 Repeat Exercise 7.47 using T flip-flops with enable.

7.50 Redesign the combination-lock machine of Table 7-14, assigning coded states in Gray-code order (A–H = 000, 001, 011, 010, 110, 111, 101, 100). Compare the cost of the resulting sum-of-products excitation equations with the ones derived in the text.

7.51 Find a 3-bit state assignment for the combination-lock machine of Table 7-14 that results in less costly excitation equations than the ones derived in the text. (*Hint:* Use the fact that inputs 1–3 are the same as inputs 4–6 in the required input sequence.)

7.52 What changes would be made to the excitation and output equations for the combination-lock machine in Section 7.4.6 as the result of performing a formal multiple-output minimization procedure (Section 4.3.8) on the five functions? You need not construct 31 product maps and go through the whole procedure; you should be able to "eyeball" the excitation and output maps in Section 7.4.6 to see what savings are possible.

7.53 Synthesize a circuit for the ambiguous state diagram in Figure 7-62. Use the state assignment in Table 7-16. Write a transition list, a transition equation for each state variable as a sum of p-terms, and simplified transition/excitation equations for a realization using D flip-flops. Determine the actual next state of the circuit, starting from the IDLE state, for each of the following input combinations on (LEFT, RIGHT, HAZ): (1,0,1), (0,1,1), (1,1,0), (1,1,1). Comment on the machine's behavior in these cases.

7.54 Suppose that for a state SA and an input combination I, an ambiguous state diagram indicates that there are two next states, SB and SC. The actual next state SD for this transition depends on the state machine's realization. If the state machine is synthesized using the $V* = \Sigma$p-terms where $V* = 1$) method to obtain transition/excitation equations for D flip-flops, what is the relationship between the coded states for SB, SC, and SD? Explain.

7.55 Repeat Exercise 7.54, assuming that the machine is synthesized using the $V*' = \Sigma$p-terms where $V* = 0$) method.

7.56 Suppose that for a state SA and an input combination I, an ambiguous state diagram does not define a next state. The actual next state SD for this transition depends on the state machine's realization. Suppose that the state machine is synthesized using the $V* = \Sigma$p-terms where $V* = 1$) method to obtain transition/excitation equations for D flip-flops. What coded state is SD? Explain.

7.57 Repeat Exercise 7.56, assuming that the machine is synthesized using the $V*' = \Sigma$p-terms where $V* = 0$) method.

7.58 Given the transition equations for a clocked synchronous state machine that is to be built using master/slave S-R flip-flops, how can the excitation equations for the S and R inputs be derived? (*Hint:* Show that any transition equation, $Qi* = \text{expr}$, can be written in the form $Qi* = Qi \cdot \text{expr1} + Qi' \cdot \text{expr2}$, and see where that leads.)

7.59 Repeat Exercise 7.58 for J-K flip-flops. How can the "don't-cares" that are possible in a J-K design be specified?

7.60 Draw a logic diagram for the output logic of the guessing-game machine in Table 7-18 using a single 74x139 dual 2-to-4 decoder. (*Hint:* Use active-low outputs.)

7.61 What does the personalized license plate in Figure 7-60 stand for? (*Hint:* It's a computer engineer's version of OTTFFSS.)

7.62 Analyze the feedback sequential circuit in Figure 7-19, assuming that the PR_L and CLR_L inputs are always 1. Derive excitation equations, construct a transition table, and analyze the transition table for critical and noncritical races. Name the states, and write a state/output table and a flow/output table. Show that the flow table performs the same function as Figure 7-85.

7.63 Draw the logic diagram for a circuit that has one feedback loop, but that is *not* a sequential circuit. That is, the circuit's output should be a function of its current input only. In order to prove your case, break the loop and analyze the circuit as if it were a feedback sequential circuit, and demonstrate that the outputs for each input combination do not depend on the "state."

7.64 A BUT *flop* may be constructed from a single NBUT gate as shown in Figure 7.64. (An NBUT *gate* is simply a BUT gate with inverted outputs; see Exercise 5.31 for the definition of a BUT gate.) Analyze the BUT flop as a feedback sequential circuit and obtain excitation equations, transition table, and flow table. Is this circuit good for anything, or is it a flop?

BUT flop
NBUT flop

7.65 Repeat Exercise 7.64 for the BUT flop in Figure 7.65.

7.66 A clever student designed the circuit in Figure 7.66 to create a BUT gate. But the circuit didn't always work correctly. Analyze the circuit and explain why.

7.67 Show that a 4-bit ones'-complement adder with end-around carry is a feedback sequential circuit.

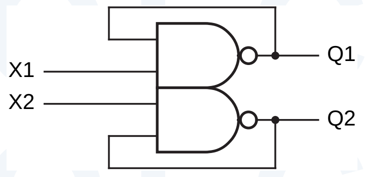


Figure X7.64

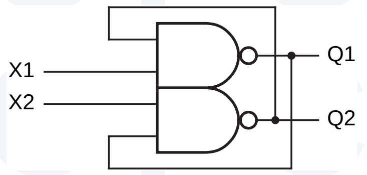


Figure X7.65

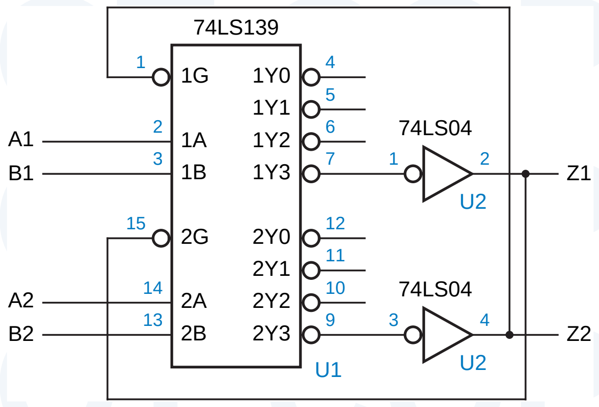


Figure X7.66

7.68 Analyze the feedback sequential circuit in Figure 7.68. Break the feedback loops, write excitation equations, and construct a transition and output table, showing the stable total states. What application might this circuit have?

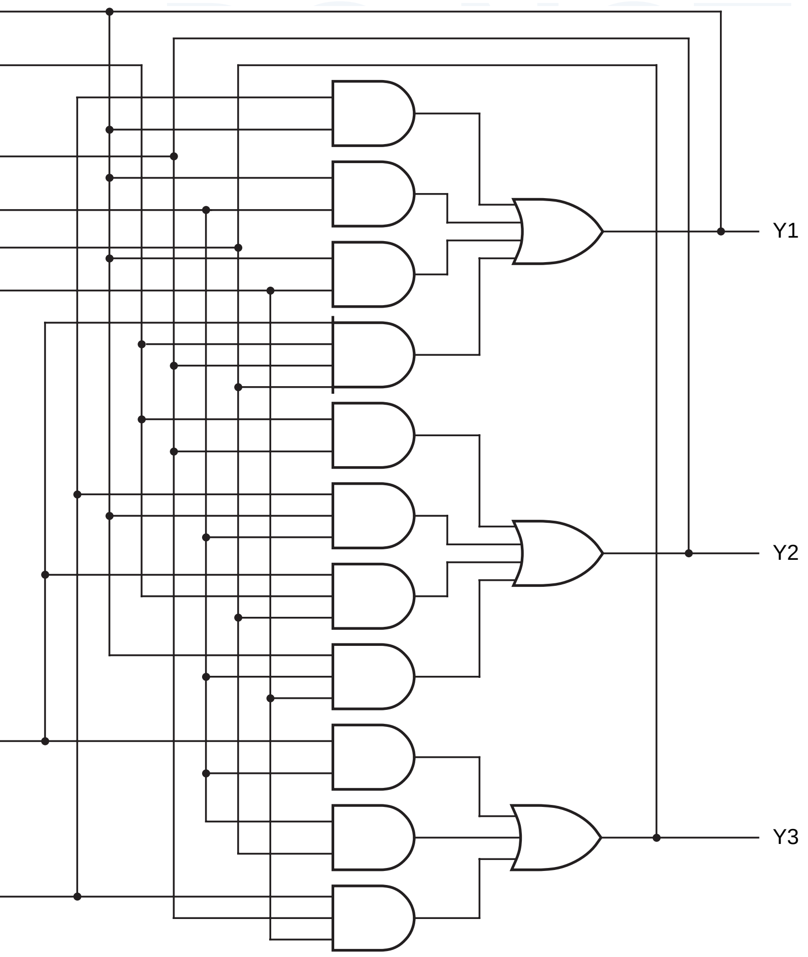
7.69 Complete the analysis of the positive-edge-triggered D flip-flop in Figure 7-86, including transition/output, state/output, and flow/output tables. Show that its behavior is equivalent to that of the D flip-flop in Figure 7-78.

7.70 We claimed in Section 7.10.1 that all single-loop feedback sequential circuits have an excitation equation of the form

$$Q* = \text{(forcing term)} + \text{(holding term)} \cdot Q$$

Why aren't there any practical circuits whose excitation equation substitutes Q′ for Q above?

7.71 Design a latch with two control inputs, C1 and C2, and three data inputs, D1, D2, and D3. The latch is to be "open" only if both control inputs are 1, and it is to store



a 1 if any of the data inputs is 1. Use hazard-free two-level sum-of-products circuits for the excitation functions.

7.72 Repeat Exercise 7.71, but minimize the number of gates required; the excitation circuits may have multiple levels of logic.

7.73 Redraw the timing diagram in Figure 7-90, showing the internal state variables of the pulse-catching circuit of Figure 7-100, assuming that it starts in state 00.

7.74 The general solution for obtaining a race-free state assignment of 2^n states using 2^{n-1} state variables yields the adjacency diagram shown in Figure 7.74 for the n = 2 case. Compare this diagram with Figure 7-97. Which is better, and why?

7.75 Design a fundamental-mode flow table for a pulse-catching circuit similar to the one described in Section 7.10.2, except that the circuit should detect both 0-to-1 and 1-to-0 transitions on P.

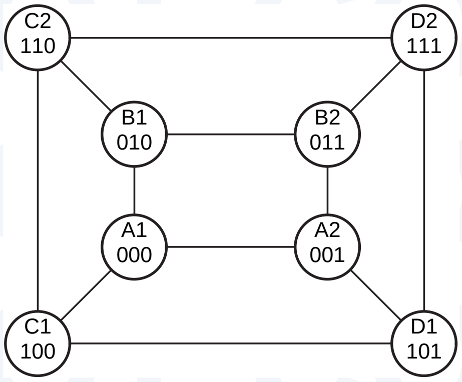


Figure X7.74

7.76 Design a fundamental-mode flow table for a double-edge-triggered D flip-flop, one that samples its inputs and changes its outputs on both edges of the clock signal.

7.77 Design a fundamental-mode flow table for a circuit with two inputs, EN and CLKIN, and a single output, CLKOUT, with the following behavior. A clock period is defined to be the interval between successive rising edges of CLKIN. If EN is asserted during an entire given clock period, then CLKOUT should be “on” during the next clock period; that is, it should be identical to CLKIN. If EN is negated during an entire given clock period, then CLKOUT should be “off” (constant 1) during the next clock period. If EN is both asserted and negated during a given clock period, then CLKOUT should be on in the next period if it had been off, and off if it had been on. After writing the fundamental-mode flow table, reduce it by combining “compatible” states if possible.

7.78 Design a circuit that meets the specifications of Exercise 7.77 using edge-triggered D flip-flops (74LS74) or JK flip-flops (74LS109) and NAND and NOR gates without feedback loops. Give a complete circuit diagram and word description of how your circuit achieves the desired behavior.

7.79 Which of the circuits of the two preceding exercises is (are) subject to metastability, and under what conditions?

7.80 For the flow table in Table 7-36, find an assignment of state variables that avoids all critical races. Additional states may be added as necessary, but use as few state variables as possible. Assign the all-0s combination to state A. Draw the adjacency diagram for the original flow table, and write the modified flow table and another adjacency diagram to support your final state-variable assignment.

7.81 Prove that the fundamental-mode flow table of any flip-flop that samples input(s) and change(s) outputs on the rising edge only of a clock signal CLK contains an essential hazard.

7.82 Locate the essential hazard(s) in the flow table for a positive-edge-triggered D flip-flop, Figure 7-85.

7.83 Identify the essential hazards, if any, in the flow table developed in Exercise 7.76.

Table 7-36

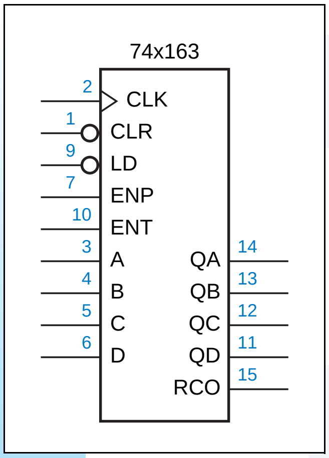
	X Y			
S	00	01	11	10
A	B	C	—	Ⓐ
B	Ⓑ	E	—	Ⓑ
C	F	Ⓒ	—	E
D	Ⓓ	F	—	B
E	D	Ⓔ	—	Ⓔ
F	Ⓕ	Ⓕ	—	A
	S*			

7.84 Identify the essential hazards, if any, in the flow table developed in Exercise 7.77.

7.85 Build a verbal flip-flop—a logical word puzzle that can be answered correctly in either of two ways depending on state. How might such a device be adapted to the political arena?

7.86 Modify the ABEL program in Table 7-27 to use an output-coded state assignment, thereby reducing the total number of PLD outputs required by one.

7.87 Finish writing the test vectors, started in Table 7-35, for the combination-lock state machine of Table 7-31. The complete set of vectors should test all of the state transitions and all of the output values for every state and input combination.



chapter 8

Sequential Logic Design Practices

The purpose of this chapter is to familiarize you with the most commonly used and dependable sequential-circuit design methods. Therefore, we will emphasize *synchronous systems*, that is, systems in which all flip-flops are clocked by the same, common clock signal. Although it's true that all the world does not march to the tick of a common clock, within the confines of a digital system or subsystem we can make it so. When we interconnect digital systems or subsystems that use different clocks, we can usually identify a limited number of asynchronous signals that need special treatment, as we'll show later.

We begin this chapter with a quick summary of sequential circuit documentation standards. After revisiting the most basic building blocks of sequential-circuit design—latches and flip-flops—we describe some of the most flexible building blocks—sequential PLDs. Next we show how counters and shift registers are realized in both MSI devices and PLDs, and show some of their applications. Finally, we show how these elements come together in synchronous systems and how the inevitable asynchronous inputs are handled.

8.1 Sequential Circuit Documentation Standards

8.1.1 General Requirements

Basic documentation standards in areas like signal naming, logic symbols, and schematic layout, which we introduced in Chapter 5, apply to digital systems as a whole and therefore to sequential circuits in particular. However, there are several ideas to highlight for system elements that are specifically “sequential”:

- *State-machine layout.* Within a logic diagram, a collection of flip-flops and combinational logic that forms a state machine should be drawn together in a logical format on the same page, so the fact that it is a state machine is obvious. (You shouldn’t have to flip pages to find the feedback path!)
- *Cascaded elements.* In a similar way, registers, counters, and shift registers that use multiple ICs should have the ICs grouped together in the schematic so that the cascading structure is obvious.
- *Flip-flops.* The symbols for individual sequential-circuit elements, especially flip-flops, should rigorously follow the appropriate drawing standards, so that the type, function, and clocking behavior of the elements are clear.
- *State-machine descriptions.* State machines should be described by state tables, state diagrams, transition lists, or text files in a state-machine description language such as ABEL or VHDL.
- *Timing diagrams.* The documentation package for sequential circuits should include timing diagrams that show the general timing assumptions and timing behavior of the circuit.
- *Timing specifications.* A sequential circuit should be accompanied by a specification of the timing requirements for proper internal operation (e.g., maximum clock frequency), as well as the requirements for any externally supplied inputs (e.g., setup- and hold-time requirements with respect to the system clock, minimum pulse widths, etc.).

8.1.2 Logic Symbols

We introduced traditional symbols for flip-flops in Section 7.2. Flip-flops are always drawn as rectangular-shaped symbols, and follow the same general guidelines as other rectangular-shaped symbols—inputs on the left, outputs on the right, bubbles for active levels, and so on. In addition, some specific guidelines apply to flip-flop symbols:

- A dynamic indicator is placed on edge-triggered clock inputs.
- A postponed-output indicator is placed on master/slave outputs that change at the end interval during which the clock is asserted.
- Asynchronous preset and clear inputs may be shown at the top and bottom of a flip-flop symbol—preset at the top and clear at the bottom.

The logic symbols for larger-scale sequential elements, such as the counters and shift register described later in this chapter, are generally drawn with all inputs, including presets and clears, on the left, and all outputs on the right. Bidirectional signals may be drawn on the left or the right, whichever is convenient.

Like individual flip-flops, larger-scale sequential elements use a dynamic indicator to indicate edge-triggered clock inputs. In "traditional" symbols, the names of the inputs and outputs give a clue of their function, but they are sometimes ambiguous. For example, two elements described later in this chapter, the 74x161 and 74x163 4-bit counters, have exactly the same traditional symbol, even though the behavior of their CLR inputs is completely different.

IEEE STANDARD SYMBOLS

IEEE standard symbols, which we show in Appendix A for all of the sequential elements in this chapter, have a rich set of notation that can provide an unambiguous definition of every signal's function.

8.1.3 State-Machine Descriptions

So far we have dealt with six different representations of state machines:

- Word descriptions
- State tables
- State diagrams
- Transition lists
- ABEL programs
- VHDL programs

You might think that having all these different ways to represent state machines is a problem—too much for you to learn! Well, they're not all that difficult to learn, but there *is* a subtle problem here.

Consider a similar problem in programming, where high-level "pseudo-code" or perhaps a flowchart might be used to document how a program works. The pseudo-code may express the programmer's intentions very well, but errors, misinterpretations, and typos can occur when the pseudo-code is translated into real code. In any creative process, inconsistencies can occur when there are multiple representations of how things work.

inconsistent state-machine representations

The same kind of inconsistencies can occur in state-machine design. A logic designer may document a machine's desired behavior with a 100%-correct hand-drawn state diagram, but you can make mistakes translating the diagram into a program, and there are *lots* of opportunities to mess up if you have to "turning the crank" manually to translate the state diagram into a state table, transition table, excitation equations, and logic diagram.

The solution to this problem is similar to the one adopted by programmers who write self-documenting code using a high-level language. The key is to select a representation that is both expressive of the designer's intentions *and* that can be translated into a physical realization using an error-free, automated process. (You don't hear many programmers screaming "Compiler bug!" when their programs don't work the first time.)

The best solution (for now, at least) is to write state-machine "programs" directly in a high-level state-machine description language like ABEL or VHDL, and to avoid alternate representations, other than general, summary word descriptions. Languages like ABEL and VHDL are easily readable and allow automatic conversion of the description into a PLD-, FPGA-, or ASIC-based realization. Some CAD tools allow state machines to be specified and synthesized using state diagrams, or even using sample timing diagrams, but these can often lead to ambiguities and unanticipated results. Thus, we'll use ABEL/VHDL approach exclusively for the rest of this book.

8.1.4 Timing Diagrams and Specifications

We showed many examples of timing diagrams in Chapters 5 and 7. In the design of synchronous systems, most timing diagrams show the relationship between the clock and various input, output, and internal signals.

Figure 8-1 shows a fairly typical timing diagram that specifies the requirements and characteristics of input and output signals in a synchronous circuit. The first line shows the system clock and its nominal timing parameters. The remaining lines show a range of delays for other signals.

For example, the second line shows that flip-flops change their outputs at some time between the rising edge of CLOCK and time t_{ffpd} afterward. External circuits that sample these signals should not do so while they are changing. The timing diagram is drawn as if the minimum value of t_{ffpd} is zero; a complete

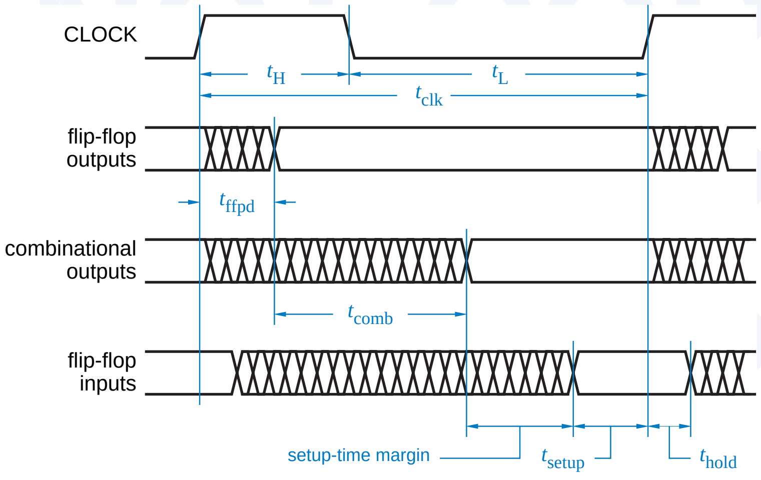


Figure 8-1
A detailed timing diagram showing propagation delays and setup and hold times with respect to the clock.

documentation package would include a timing table indicating the actual minimum, typical, and maximum values of t_{ffpd} and all other timing parameters.

The third line of the timing diagram shows the additional time, t_{comb}, required for the flip-flop output changes to propagate through combinational logic elements, such as flip-flop excitation logic. The excitation inputs of flip-flops and other clocked devices require a setup time of $\mathrm{t}_{\mathrm{setup}}$, as shown in the fourth line. For proper circuit operation we must have $t_{\mathrm{clk}} - t_{\mathrm{ffpd}} - t_{\mathrm{comb}} > t_{\mathrm{setup}}$.

timing margin

Timing margins indicate how much "worse than worst-case" the individual components of a circuit can be without causing the circuit to fail. Well-designed systems have positive, nonzero timing margins to allow for unexpected circumstances (marginal components, brown-outs, engineering errors, etc.) and clock skew (Section 8.8.1).

setup-time margin

The value $t_{\mathrm{clk}} - t_{\mathrm{ffpd(max)}} - t_{\mathrm{comb(max)}} - t_{\mathrm{setup}}$ is called the *setup-time margin;* if this is negative, the circuit won't work. Note that *maximum* propagation delays are used to calculate setup-time margin. Another timing margin involves the hold-time requirement t_{hold}; the sum of the *minimum* values of t_{ffpd} and t_{comb} must be greater than t_{hold}, and the *hold-time margin* is $t_{\mathrm{ffpd(min)}} + t_{\mathrm{comb(min)}} - t_{\mathrm{hold}}$.

hold-time margin

The timing diagram in Figure 8-1 does not show the timing differences between different flip-flop inputs or combinational-logic signals, even though such differences exist in most circuits. For example, one flip-flop's Q output may be connected directly to another flip-flop's D input, so that t_{comb} for that path is zero, while another's may go the ripple-carry path of a 32-bit adder before reaching a flip-flop input. When proper synchronous design methodology is used, these relative timings are not critical, since none of these signals affect the state of the circuit until a clock edge occurs. You merely have to find the longest delay path in one clock period to determine whether the circuit will work. However, you may have to analyze several different paths in order to find the worst-case one.

Another, perhaps more common, type of timing diagram shows only functional behavior and is not concerned with actual delay amounts; an example is shown in Figure 8-2. Here, the clock is "perfect." Whether to show signal changes as vertical or slanted lines is strictly a matter of personal taste in this and all other timing diagrams, unless rise and fall times must be explicitly indicated. Clock transitions are shown as vertical lines in this and other figures in keeping with the idea that the clock is a "perfect" reference signal.

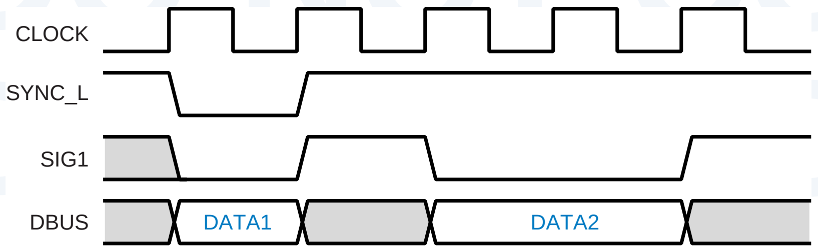


Figure 8-2
Functional timing of a synchronous circuit.

NOTHING'S PERFECT

In reality, there's no such thing as a perfect clock signal. One imperfection that most designers of high-speed digital circuits have to deal with is "clock skew." As we show in Section 8.8.1, a given clock edge arrives at different circuit inputs at different times because of differences in wiring delays, loading, and other effects.

Another imperfection, a bit beyond the scope of this text, is "clock jitter." A 10-MHz clock does not have a period of exactly 100 ns on every cycle—it may be 100.05 ns in one cycle, and 99.95 ns in the next. This is not a big deal in such a slow circuit, but in a 500-MHz circuit, the same 0.1 ns of jitter eats up 5% of the 2-ns timing budget. And the jitter in some clock sources is even higher!

The other signals in Figure 8-2 may be flip-flop outputs, combinational outputs, or flip-flop inputs. Shading is used to indicate "don't-care" signal values; cross-hatching as in Figure 8-1 on the preceding page could be used instead. All of the signals are shown to change immediately after the clock edge. In reality, the outputs change sometime later, and inputs may change just barely before the next clock edge. However, "lining up" everything on the clock edge allows the timing diagram to display more clearly which functions are performed during each clock period. Signals that are lined up with the clock are simply understood to change sometime *after* the clock edge, with timing that meets the setup- and hold-time requirements of the circuit. Many timing diagrams of this type appear in this chapter.

Table 8-1 shows manufacturer's timing parameters for commonly used flip-flops, registers, and latches in CMOS and TTL. "Typical" values are for

Table 8-1 Propagation delay in ns of selected CMOS flip-flops, registers, and latches.

		74HCT		*74AHCT*		*74FCT*		*74LS*	
Part	***Parameter***	***Typ.***	***Max.***	***Typ.***	***Max.***	***Min.***	***Max.***	***Typ.***	***Max.***
'74	t_{pd}, CLK↑ to Q or $\overline{\text{Q}}$	35	44	6.3	10			25	40
	t_{pd}, $\overline{\text{PR}}$↓ or $\overline{\text{CLR}}$↓ to Q or $\overline{\text{Q}}$	40	50	8.1	13			25	40
	t_s, D to CLK↑	12	15		5				20
	t_h, D from CLK↑	3	3		0				5
	t_{rec}, CLK↑ from $\overline{\text{PR}}$↑ or $\overline{\text{CLR}}$↑	6	8		3.5				
	t_w, CLK low or high	18	23		5				25
	t_w, $\overline{\text{PR}}$ or $\overline{\text{CLR}}$ low	16	20		5				25
'174	t_{pd}, CLK↑ to Q	40	50	6.3	10			21	30
	t_{pd}, $\overline{\text{CLR}}$↓ to Q	44	55	8.1	13			23	35
	t_s, D to CLK↑	16	20		5				20
	t_h, D from CLK↑	5	5		0				5
	t_{rec}, CLK↑ from $\overline{\text{CLR}}$↑	12	15		3.5				25
	t_w, CLK low or high	20	25		5				20
	t_w, $\overline{\text{CLR}}$ low	25	31		5				20

Table 8-1 (continued) Propagation delay in ns of selected CMOS flip-flops, registers, and latches.

		74HCT		74AHCT		74FCT		74LS	
Part	**Parameter**	**Typ.**	**Max.**	**Typ.**	**Max.**	**Min.**	**Max.**	**Typ.**	**Max.**
’175	t_{pd}, CLK↑ to Q or $\overline{\text{Q}}$	33	41					21	30
	t_{pd}, $\overline{\text{CLR}}$↓ to Q or $\overline{\text{Q}}$	35	44					23	35
	t_s, D to CLK↑	20	25						20
	t_h, D from CLK↑	5	5						5
	t_{rec}, CLK↑ from $\overline{\text{CLR}}$↑	5	5						25
	t_w, CLK low or high	20	25						20
	t_w, $\overline{\text{CLR}}$ low	20	25						20
’273	t_{pd}, CLK↑ to Q	30	38	6.8	11	2	7.2	18	27
	t_{pd}, $\overline{\text{CLR}}$↓ to Q	32	40	8.5	12.6	2	7.2	18	27
	t_s, D to CLK↑	12	15		5		2		20
	t_h, D from CLK↑	3	3		0		1.5		5
	t_{rec}, CLK↑ from $\overline{\text{CLR}}$↑	10	13		2.5		2		25
	t_w, CLK low or high	20	25		6.5		4		20
	t_w, $\overline{\text{CLR}}$ low	12	15		6		5		20
’373	t_{pd}, C↑ to Q	35	44	8.5	14.5	2	8.5	24	36
	t_{pd}, D to Q	32	40	5.9	10.5	1.5	5.2	18	27
	t_s, D to C↓	10	13		1.5		2		0
	t_h, D from C↓	5	5		3.5		1.5		10
	t_{pHZ}, $\overline{\text{OE}}$ to Q	35	44		12	1.5	6.5	12	20
	t_{pLZ}, $\overline{\text{OE}}$ to Q	35	44		12	1.5	6.5	16	25
	t_{pZH}, $\overline{\text{OE}}$ to Q	35	44		13.5	1.5	5.5	16	28
	t_{pZL}, $\overline{\text{OE}}$ to Q	35	44		13.5	1.5	5.5	22	36
	t_w, C low or high	16	20		6.5		5		15
’374	t_{pd}, CLK↑ to Q	33	41	6.4	11.5	2	6.5	22	34
	t_s, D to CLK↑	12	15		2.5		2		20
	t_h, D from CLK↑	5	5		2.5		1.5		0
	t_{pHZ}, $\overline{\text{OE}}$ to Q	28	35		12	1.5	6.5		18
	t_{pLZ}, $\overline{\text{OE}}$ to Q	28	35		12	1.5	6.5		24
	t_{pZH}, $\overline{\text{OE}}$ to Q	30	38		12.5	1.5	5.5		28
	t_{pZL}, $\overline{\text{OE}}$ to Q	30	38		12.5	1.5	5.5		36
	t_w, CLK low or high	16	20		6.5		5		15
’377	t_{pd}, CLK↑ to Q	38	48			2	7.2	18	27
	t_s, D to CLK↑	12	15				2		20
	t_h, D from CLK↑	3	3				1.5		5
	t_s, $\overline{\text{EN}}$ to CLK↑	12	15				2		25
	t_h, $\overline{\text{EN}}$ from CLK↑	5	5				1.5		5
	t_w, CLK low or high	20	25				6		20

devices operating at 25ºC but, depending on the logic family, they could be for a typical part and nominal power-supply voltage, or they could be for a worst-case part at worst-case supply voltage. "Maximum" values are generally valid over the commercial operating range of voltage and temperature, except TTL values, which are specified at 25ºC. Also note that the "maximum" values of t_{s}, t_{h}, t_{rec}, or t_{w} are the maximum values of the *minimum* setup time, hold time, recovery time, or pulse width that the specified part will exhibit.

Different manufacturers may use slightly different definitions for the same timing parameters, and they may specify different numbers for the same part. A given manufacturer may even use different definitions for different families or part numbers in the same family. Thus, all of the specifications in Table 8-1 are merely representative; for exact numbers *and* their definitions, you must consult the data sheet for the particular part and manufacturer.

8.2 Latches and Flip-Flops

8.2.1 SSI Latches and Flip-Flops

Several different types of discrete latches and flip-flops are available as SSI parts. These devices are sometimes used in the design of state machines and "unstructured" sequential circuits that don't fall into the categories of shift registers, counters, and other sequential MSI functions presented later in this chapter. However, SSI latches and flip-flops have been eliminated to a large extent in modern designs as their functions are embedded in PLDs and FPGAs. Nevertheless, a handful of these discrete building blocks still appear in many digital systems, so it's important to be familiar with them.

74x375

Figure 8-3 shows the pinouts for several SSI sequential devices. The only latch in the figure is the *74x375*, which contains four D latches, similar in function to the "generic" D latches described in Section 7.2.4. Because of pin limitations, the latches are arranged in pairs with a common C control line for each pair.

74x74

Among the devices in Figure 8-3, the most important is the *74x74*, which contains two independent positive-edge-triggered D flip-flops with preset and clear inputs. We described the functional operation, timing, and internal structure of edge-triggered D flip-flops in general, and the 74x74 in particular, in Section 7.2.5. Besides the 74x74's use in "random" sequential circuits, fast versions of the part, such as the *74F74* and *74ACT74*, find application in synchronizers for asynchronous input signals, as discussed in Section 8.9.

74F74
74ACT74
74x109

The *74x109* is a positive-edge-triggered J-$\overline{\mathrm{K}}$ flip-flop with an active-low K input (named $\overline{\mathrm{K}}$ or K_L). We discussed the internal structure of the '109 in Section 7.2.10. Another J-K flip-flop is the *74x112*, which has an active-low clock input.

74x112

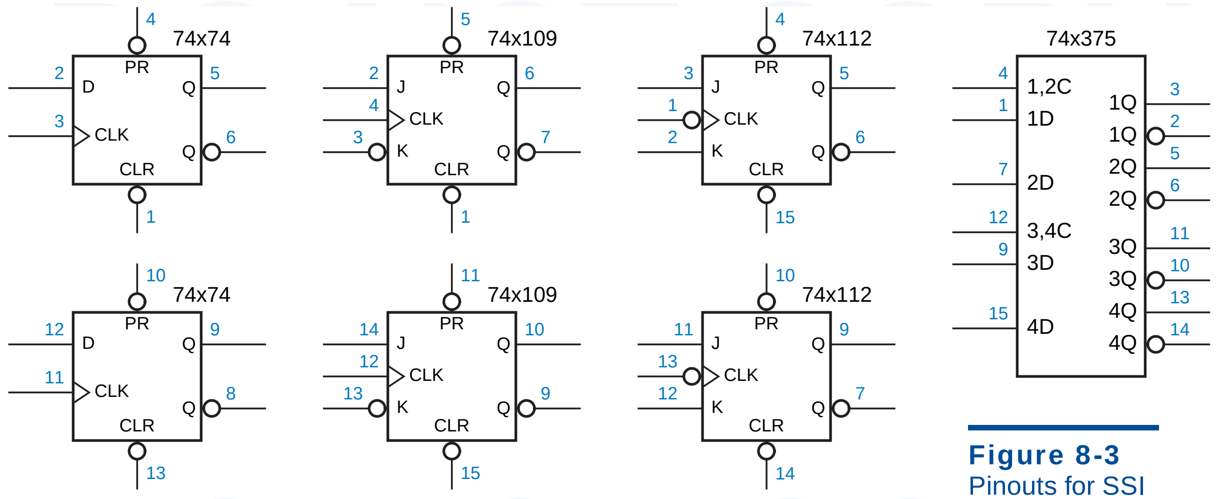


Figure 8-3
Pinouts for SSI latches and flip-flops.

*8.2.2 Switch Debouncing

A common application of simple bistables and latches is switch debouncing. We're all familiar with electrical switches from experience with lights, garbage disposals, and other appliances. Switches connected to sources of constant logic 0 and 1 are often used in digital systems to supply "user inputs." However, in digital logic applications we must consider another aspect of switch operation, the time dimension. A simple make or break operation, which occurs instantly as far as we slow-moving humans are concerned, actually has several phases that are discernible by high-speed digital logic.

Figure 8-4(a) shows how a single-pole, single-throw (SPST) switch might be used to generate a single logic input. A pull-up resistor provides a logic-1 value when the switch is opened, and the switch contact is tied to ground to provide a logic-1 value when the switch is pushed.

As shown in (b), it takes a while after a push for the wiper to hit the bottom contact. Once it hits, it doesn't stay there for long; it bounces a few times before finally settling. The result is that several transitions are seen on the SW_L and DSW logic signals for each single switch push. This behavior is called *contact bounce*. Typical switches bounce for 10–20 ms, a very long time compared to the switching speeds of logic gates.

contact bounce

Contact bounce may or may not be a problem, depending on the switch application. For example, some computers have configuration information specified by small switches, called *DIP switches* because they are the same size as a dual in-line package (DIP). Since DIP switches are normally changed only when the computer is inactive, there's no problem. Contact bounce *is* a problem

DIP switch

* Throughout this book, optional sections are marked with an asterisk.

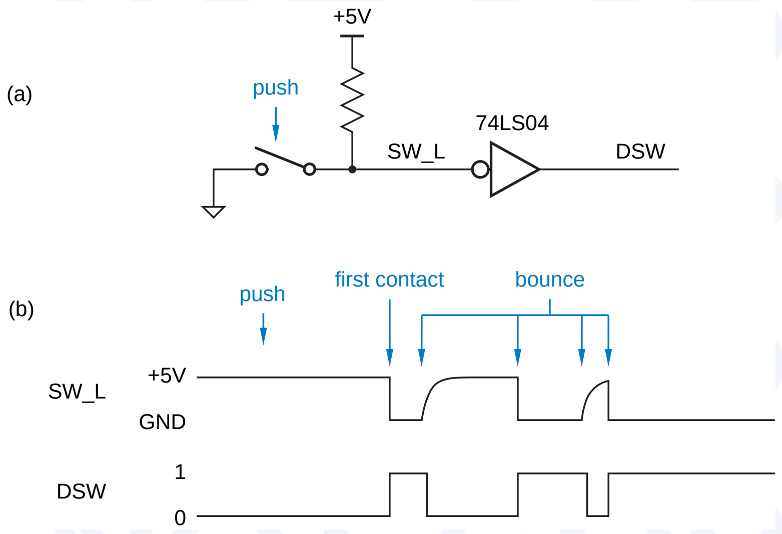


Figure 8-4
Switch input without debouncing.

if a switch is being used to count or signal some event (e.g., laps in a race). Then we must provide a circuit (or, in microprocessor-based systems, software) to *debounce* the switch—to provide just one signal change or pulse for each external event.

debounce

*8.2.3 The Simplest Switch Debouncer

Switch debouncing is a good application for the simplest sequential circuit, the bistable element of Section 7.1, which can be used as shown in Figure 8-5. This circuit uses a single-pole, double-throw (SPDT) switch. The switch contacts and wiper have a “break before make” behavior, so the wiper terminal is “floating” at some time halfway through the switch depression. i

Before the button is pushed, the top contact holds SW at 0 V, a valid logic 0, and the top inverter produces a logic 1 on SW_L and on the bottom contact. When the button is pushed and contact is broken, feedback in the bistable holds SW at V_{OL} (≤ 0.5 V for LS-TTL), still a valid logic 0.

Next, when the wiper hits the bottom contact, the circuit operates quite unconventionally for a moment. The top inverter in the bistable is trying to maintain a logic 1 on the SW_L signal; the top transistor in its totem-pole output is “on” and connecting SW_L through a small resistance to +5 V. Suddenly, the switch contact makes a metallic connection of SW_L to ground, 0.0 V. Not surprisingly, the switch contact wins.

A short time later (30 ns for the 74LS04), the forced logic 0 on SW_L propagates through the two inverters of the bistable, so that the top inverter gives up its vain attempt to drive a 1, and instead drives a logic 0 onto SW_L. At this point, the top inverter output is no longer shorted to ground, and feedback in the bistable maintains the logic 0 on SW_L even if the wiper bounces off the bottom contact, as it does. (It does not bounce far enough to touch the top contact again.)

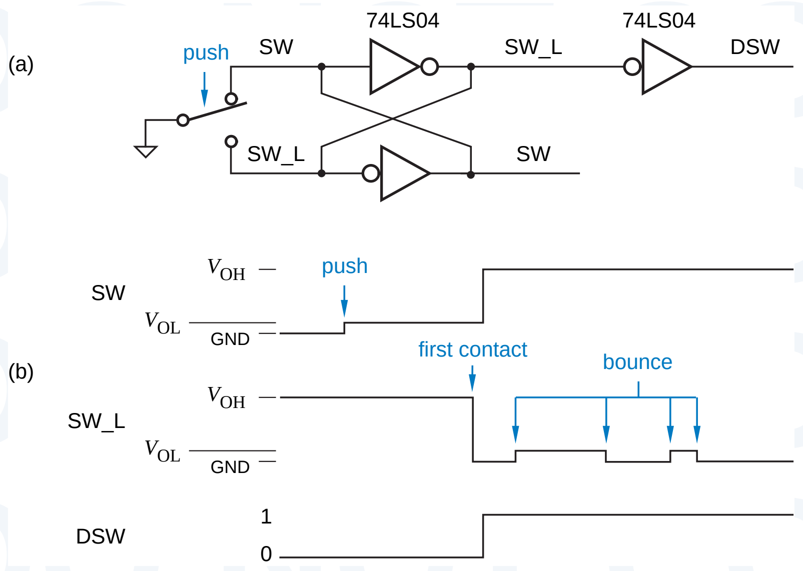


Figure 8-5 Switch input using a bistable for debouncing

Advantages of this circuit compared to other debouncing approaches are that it has a low chip count (one-third of a 74LS04), no pull-up resistors are required, and both polarities of the input signal (active-high and active-low) are produced. In situations where momentarily shorting gate outputs must be avoided, a similar circuit can be designed using a $\overline{\text{S}}$-$\overline{\text{R}}$ latch and pull-up resistors, as suggested in Figure 8-6.

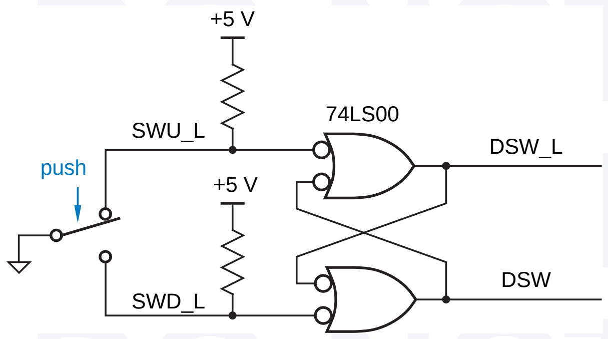


Figure 8-6 Switch input using an $\overline{\text{S}}$-$\overline{\text{R}}$ latch for debouncing.

WHERE WIMPY WORKS WELL

The circuit in Figure 8-5, while elegant, should not be used with high-speed CMOS devices, like the 74ACT04, whose outputs are capable of sourcing large amounts of current in the HIGH state. While shorting such outputs to ground momentarily will not cause any damage, it will generate a noise pulse on power and ground signals that may trigger improper operation of the circuit elsewhere. The debouncing circuit in the figure works well with wimpy logic families like HCT and LS-TTL.

*8.2.4 Bus Holder Circuit

In Sections 3.7.3 and 5.6 we described three-state outputs and how they are tied together to create three-state buses. At any time, at most one output can drive the bus; sometimes, no output is driving the bus, and the bus is “floating.” When high-speed CMOS inputs are connected to a bus that is left floating for a long time (in the fastest circuits, more than a clock tick or two), bad things can happen. In particular, noise, crosstalk, and other effects can drive the high-impedance floating bus signals to a voltage level near the CMOS devices’ input switching threshold, which in turn allows excessive current to flow in the device outputs. For this reason, it is desirable and customary to provide pull-up resistors that quickly pull a floating bus to a valid HIGH logic level.

Pull-up resistors aren’t all goodness—they cost money and they occupy valuable printed-circuit-board real estate. A big problem they have in very high-speed circuits is the choice of resistance value. If the resistance is too high, when a bus goes from LOW to floating, the transition from LOW to pulled-up (HIGH) will be slow due to the high *RC* time constant, and input levels may spend too much time near the switching threshold. If the pull-up resistance is too low, devices trying to pull the bus LOW will have to sink too much current.

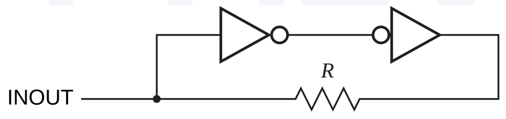
bus holder circuit

The solution to this problem is to eliminate pull-up resistors in favor of an active *bus holder circuit*, shown in Figure 8-7. This is nothing but a bistable with a resistor *R* in one leg of the feedback loop. The bus holder’s INOUT signal is connected to the three-state bus line which is to be held. When the three-state output currently driving the line LOW or HIGH changes to floating, the bus holder’s right-hand inverter holds the line in its current state. When a three-state output tries to change the line from LOW to HIGH or vice versa, it must source or sink a small amount of additional current through *R* to overcome the bus holder. This additional current flow persists only for the short time that it takes for the bistable to flip into its other stable state.

The choice of the value of *R* in the bus holder is a compromise between low override current (high *R*) and good noise immunity on the held bus line (low *R*). A typical example, bus holder circuits in the 3.3-V CMOS LVC family specify a maximum override current of 500 μA, implying $R \approx 3.3 / 0.0005 = 6.6\text{K}\Omega$.

Bus holder circuits are often built into another MSI device, such as an octal CMOS bus driver or transceiver. They require no extra pins and require very little chip area, so they are essentially free. And there’s no real problem in having multiple (*n*) bus holders on the same signal line, as long as the bus drivers can provide *n* times the override current for a few nanoseconds during switching. Note that bus holders normally are not effective on buses that have TTL inputs attached to them (see Exercise 8.14).

Figure 8-7
Bus holder circuit.



8.2.5 Multibit Registers and Latches

A collection of two or more D flip-flops with a common clock input is called a *register*. Registers are often used to store a collection of related bits, such as a byte of data in a computer. However, a single register can also be used to store unrelated bits of data or control information; the only real constraint is that all of the bits are stored using the same clock signal.

register

Figure 8-8 shows the logic diagram and logic symbol for a commonly used MSI register, the *74x175*. The 74x175 contains four edge-triggered D flip-flops with a common clock and asynchronous clear inputs. It provides both active-high and active-low outputs at the external pins of the device.

74x175

The individual flip-flops in a '175 are negative-edge triggered, as indicated by the inversion bubbles on their CLK inputs. However, the circuit also contains an inverter that makes the flip-flops positive-edge triggered with respect to the device's external CLK input pin. The common, active-low, clear signal (CLR_L) is connected to the asynchronous clear inputs of all four flip-flops. Both CLK and CLR_L are buffered before fanning out to the four flip-flops, so that a device driving one of these inputs sees only one unit load instead of four. This is especially important if a common clock or clear signal must drive many such registers.

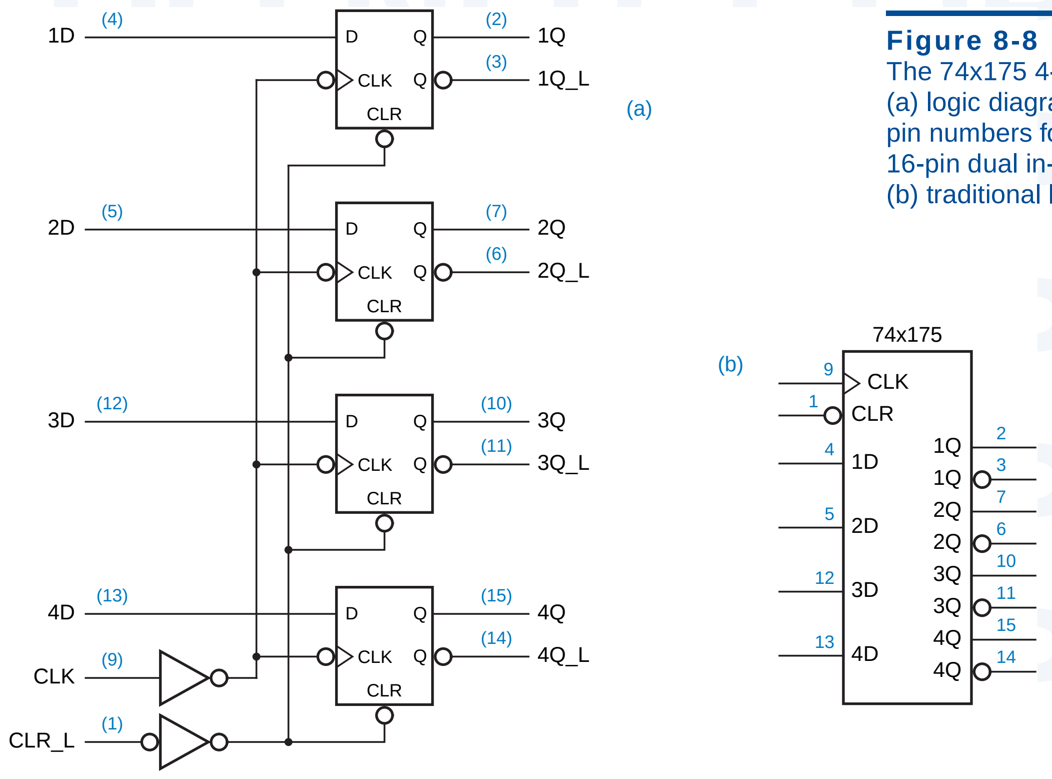


Figure 8-8 The 74x175 4-bit register: (a) logic diagram, including pin numbers for a standard 16-pin dual in-line package; (b) traditional logic symbol.

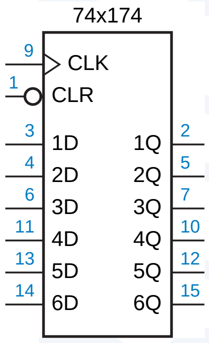


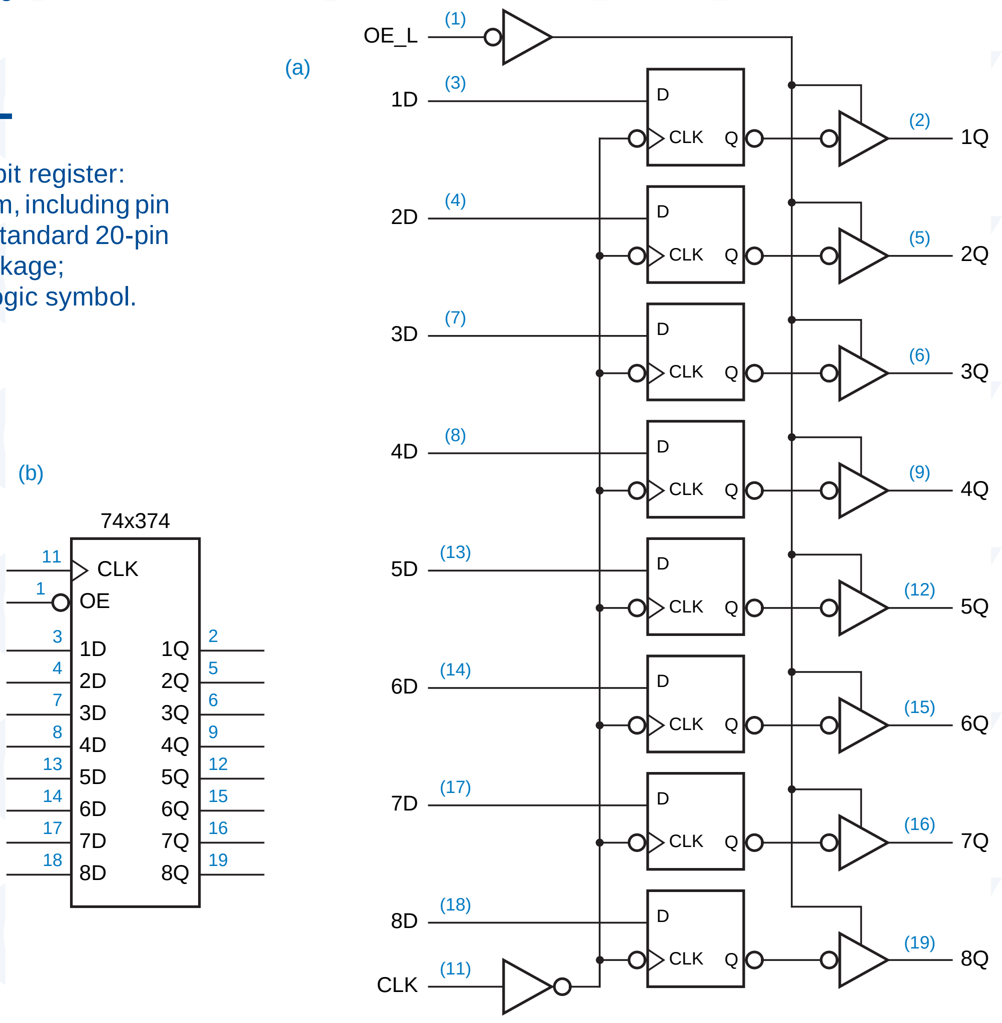
Figure 8-9
Logic symbol for the 74x174 6-bit register.

The logic symbol for the *74x174*, 6-bit register is shown in Figure 8-9. The internal structure of this device is similar to the 74x175's, is similar, except that it eliminates the active-low outputs and provides two more flip-flops instead.

Many digital systems, including computers, telecommunications devices, and stereo equipment, process information 8, 16, or 32 bits at a time; as a result, ICs that handle eight bits are very popular. One such MSI IC is the *74x374* octal edge-triggered D flip-flop, also known simply as an 8-bit register. (Once again, "octal" means that the device has eight sections.)

As shown in Figure 8-10, the 74x374 contains eight edge-triggered D flip-flops that all sample their inputs and change their outputs on the rising edge of a common CLK input. Each flip-flop output drives a three-state buffer that in turn drives an active-high output. All of the three-state buffers are enabled by a

Figure 8-10
The 74x374 8-bit register:
(a) logic diagram, including pin numbers for a standard 20-pin dual in-line package;
(b) traditional logic symbol.



common, active-low OE_L (output enable) input. As in the other registers that we've studied, the control inputs (CLK and OE_L) are buffered so that they present only one unit load to a device that drives them.

One variation of the 74x374 is the *74x373*, whose symbol is shown in Figure 8-11. The '373 uses D latches instead of edge-triggered flip-flops. Therefore, its outputs follow the corresponding inputs whenever C is asserted, and latch the last input values when C is negated. Another variation is the *74x273*, shown in Figure 8-12. This octal register has non-three-state outputs and no OE_L input; instead it uses pin 1 for an asynchronous clear input CLR_L.

The *74x377*, whose symbol is shown in Figure 8-13(a), is an edge-triggered register like the '374, but it does not have three-state outputs. Instead, pin 1 is used as an active-low clock enable input EN_L. If EN_L is asserted (LOW) at the rising edge of the clock, then the flip-flops are loaded from the data inputs; otherwise, they retain their present values, as shown logically in (b).

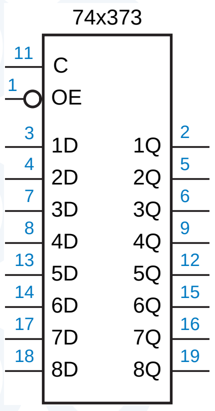


Figure 8-11 Logic symbol for the 74x373 8-bit latch.

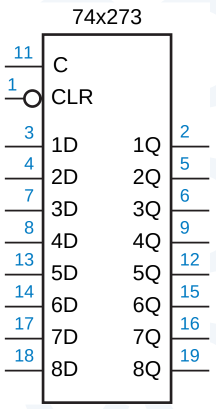
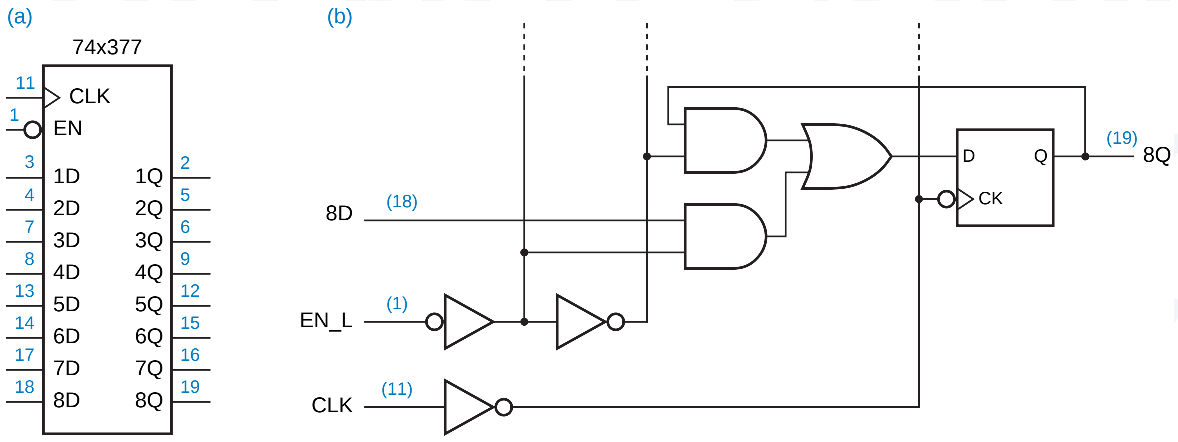


Figure 8-12 Logic symbol for the 74x273 8-bit register.

Figure 8-13 The 74x377 8-bit register with gated clock: (a) logic symbol; (b) logical behavior of one bit.



High pin-count surface-mount packaging supports even wider registers, drivers, and transceivers. Most common are 16-bit devices, but there are also devices with 18 bits (for byte parity) and 32 bits. Also, the larger packages can offer more control functions, such as clear, clock enable, multiple output enables, and even a choice of latching vs. registered behavior all in one device.

8.2.6 Registers and Latches in ABEL and PLDs

As we showed in Section 7.11, registers are very easy to specify in ABEL. For example, Table 7-33 on page 543 showed an ABEL program for an 8-bit register with enable. Obviously, ABEL allows the functions performed at the D inputs of register to be customized in almost any way desired, limited only by the number of inputs and product terms in the targeted PLD. We describe sequential PLDs in Section 8.3.

With most sequential PLDs, few if any customizations can be applied to a register's clock input (e.g, polarity choice) or to the asynchronous inputs (e.g., different preset conditions for different bits). However, ABEL does provide appropriate syntax to apply these customizations in devices that support them, as described in Section 7.11.1.

Very few PLDs have latches built in; edge-triggered registers are much more common, and generally more useful. However, you can also synthesize a latch using combinational logic and feedback. For example, the excitation equation for an S-R latch is

$$Q* = S + R' \cdot Q$$

Thus, you could build an S-R latch using one combinational output of a PLD, using the ABEL equation "`Q = S # !R & Q`." Furthermore, the S and R signals above could be replaced with more complex logic functions of the PLD's inputs, limited only by the availability of product terms (seven per output in a 16V8C or 16L8) to realize the final excitation equation. The feedback loop can be created only when Q is assigned to a bidirectional pin (in a 16V8C or 16L8, pins IO2–IO7, not O1 or O8). Also, the output pin must be continuously output-enabled; otherwise, the feedback loop would be broken and the latch's state lost.

Probably the handiest latch to build out of a combinational PLD is a D latch. The basic excitation equation for a D latch is

$$Q* = C \cdot D + C' \cdot Q$$

However, we showed in Section 7.10.1 that this equation contains a static hazard, and the corresponding circuit does not latch data reliably. To build a reliable D latch, we must include a consensus term in the excitation equation:

$$Q* = C \cdot D + C' \cdot Q + D \cdot Q$$

The D input in this equation may be replaced with a more complicated expression, but the equation's structure remains the same:

$$Q* = C \cdot \mathit{expression} + C' \cdot Q + \mathit{expression} \cdot Q$$

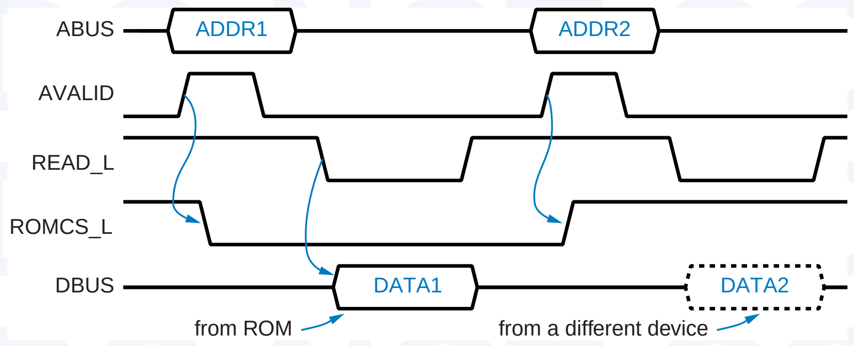


Figure 8-14
Timing diagram for a microprocessor read operation.

It is also possible to use a more complex expression for the C input, as we showed in Section 7.10.1. In any case, it is very important for the consensus term to be included in the PLD realization. The compiler can work against you in this case, since its minimization step will find that the consensus term is redundant and remove it.

`retain` property

Some versions of the ABEL compiler let you prevent elimination of consensus terms by including a keyword "`retain`" in the property list of the `istype` declaration for any output which is not to be minimized. In other versions, your only choice is to turn off minimization for the entire design.

Probably the most common use of a PLD-based latch is to simultaneously decode and latch addresses in order to select memory and I/O devices in microprocessor systems. Figure 8-14 is a timing diagram for this function in a typical system. The microprocessor selects a device and a location within the device by placing an address on its address bus (ABUS) and asserting an "address valid" signal (AVALID). A short time later, it asserts a read signal (READ_L), and the selected device responds by placing data on the data bus (DBUS).

Notice that the address does not stay valid on ABUS for the entire operation. The microprocessor bus protocol expects the address to be latched using AVALID as an enable, then decoded, as shown in Figure 8-15. The decoder selects different devices to be enabled or "chip-selected" according to the high-order bits of the address (the 12 high-order bits in this example). The low-order bits are used to address individual locations of a selected device.

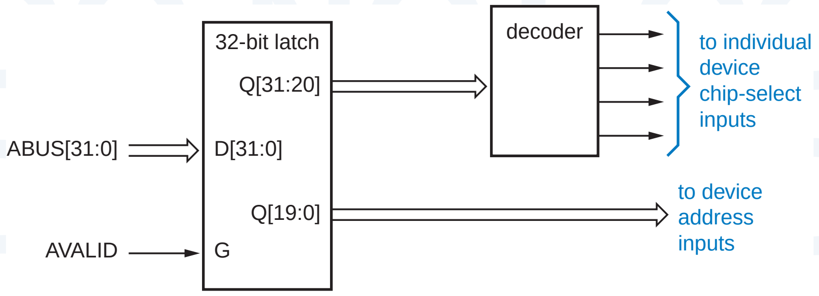


Figure 8-15
Microprocessor address latching and decoding circuit.

WHY A LATCH? The microprocessor bus protocol in Figure 8-14 raises several questions:

- Why not keep the address valid on ABUS for the entire operation? In a real system using this protocol, the functions of ABUS and DBUS are combined (multiplexed) onto one three-state bus to save pins and wires.
- Why not use AVALID as the clock input to a positive-edge-triggered register to capture the address? There isn't enough setup time; in a real system, the address may first be valid at or slightly after the rising edge of AVALID.
- OK, so why not use AVALID to clock a negative-edge-triggered register? This works, but the latched outputs are available sooner; valid values on ABUS flow through a latch immediately, without waiting for the falling clock edge. This relaxes the access-time requirements of memories and other devices driven by the latched outputs.

Using a PLD, the latching and decoding functions for the high-order bits can be combined into a single device, yielding the block diagram in Figure 8-16. Compared with Figure 8-15, the "latching decoder" saves devices and pins, and may produce a valid chip-select output more quickly (see Exercise 8.1).

Table 8-2 is an ABEL program for the latching decoder. Since it operates on only the high-order bits `ABUS[31..20]`, it can decode addresses only in 1-Mbyte or larger chunks (2^{20} = 1M). A read-only memory (ROM) is located in the highest 1-Mbyte chunk, addresses `0xfff00000`–`0xffffffff`, and is selected by `ROMCS`. Three 16-Mbyte banks of read/write memory (RAM) are located at lower addresses, starting at addresses `0x00000000`, `0x00100000`, and `0x00200000`, respectively. Notice how don't-cares are used in the definitions of the RAM bank address ranges to decode a chunk larger than 1 Mbyte. Other approaches to these definitions are also possible (e.g., see Exercise 8.2).

The equations in Table 8-2 for the chip-select outputs follow the D-latch template that we gave on page 576. The expressions that select a device, such as "`ABUS==ROM`," each generate a single product term, and each equation generates three product terms. Notice the use of the "`retain`" property in the pin declarations to prevent the compiler from optimizing away the consensus terms.

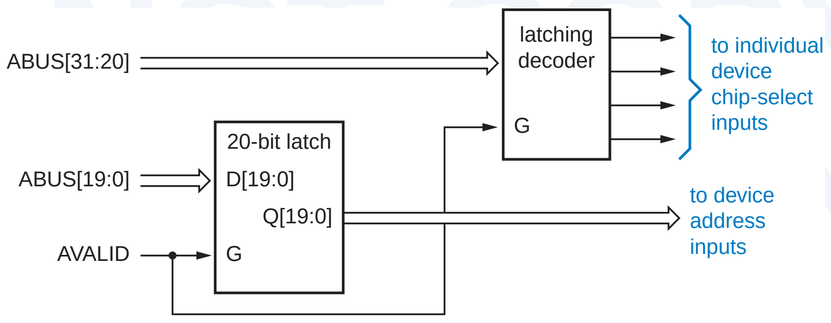


Figure 8-16
Using a combined address latching and decoding circuit.

```
module latchdec
title 'Latching Microprocessor Address Decoder'

" Inputs
AVALID, ABUS31..ABUS20          pin;
" Latched and decoded outputs
ROMCS, RAMCS0, RAMCS1, RAMCS2    pin istype 'com,retain';

ABUS = [ABUS31..ABUS20];
ROM = ^hFFF;
RAMBANK0 = [0,0,0,0,0,0,0,0,.X.,.X.,.X.,.X.];
RAMBANK1 = [0,0,0,0,0,0,0,1,.X.,.X.,.X.,.X.];
RAMBANK2 = [0,0,0,0,0,0,1,0,.X.,.X.,.X.,.X.];

equations

ROMCS  = AVALID & (ABUS==ROM) # !AVALID & ROMCS
       # (ABUS==ROM) & ROMCS;
RAMCS0 = AVALID & (ABUS==RAMBANK0) # !AVALID & RAMCS0
       # (ABUS==RAMBANK0) & RAMCS0;
RAMCS1 = AVALID & (ABUS==RAMBANK1) # !AVALID & RAMCS1
       # (ABUS==RAMBANK1) & RAMCS1;
RAMCS2 = AVALID & (ABUS==RAMBANK2) # !AVALID & RAMCS2
       # (ABUS==RAMBANK2) & RAMCS2;

end latchdec
```

Table 8-2 ABEL program for a latching address decoder.

After seeing how easy it is to build S-R and D latches using combinational PLDs, you might be tempted to go further and try to build an edge-triggered D flip-flop. Although this is possible, it is expensive because an edge-triggered flip-flop has four internal states and thus two feedback loops, consuming two PLD outputs. Furthermore, the setup and hold times and propagation delays of such a flip-flop would be quite poor compared to those of a discrete flip-flop in the same technology. Finally, as we discussed in Section 7.10.6, the flow tables of all edge-triggered flip-flops contain essential hazards, which can be masked only by controlling path delays, difficult in a PLD-based design.

8.2.7 Registers and Latches in VHDL

Register and latch circuits can be specified using structural VHDL. For example, Table 8-3 is a structural VHDL program corresponding to the D latch circuit of Figure 7-12 on page 443. However, writing structural programs is not really our motivation for using VHDL; our goal is to use behavioral programs to model the operation of circuits more intuitively.

Table 8-4 is a process-based behavioral architecture for the D latch that requires, in effect, just one line of code to describe the latch's behavior. Note that the VHDL compiler "infers" a latch from this description—since the code

Table 8-3 VHDL structural program for the D latch in Figure 7-12.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdlatch is
  port (D, C: in STD_LOGIC;
        Q, QN: buffer STD_LOGIC );
end Vdlatch;

architecture Vdlatch_s of Vdlatch is
  signal DN, SN, RN: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component nand2b port (I0, I1: in STD_LOGIC; O: buffer STD_LOGIC ); end component;
begin
  U1: inv port map (D,DN);
  U2: nand2b port map (D,C,SN);
  U3: nand2b port map (C,DN,RN);
  U4: nand2b port map (SN,QN,Q);
  U5: nand2b port map (Q,RN,QN);
end Vdlatch_s;
```

Table 8-4 VHDL behavioral architecture for a D latch.

```
architecture Vdlatch_b of Vdlatch is
begin
process(C, D, Q)
  begin
    if (C='1') then Q <= D; end if;
    QN <= not Q;
  end process;
end Vdlatch_b;
```

inferred latch

doesn't say what to do if C is not 1, the compiler creates an *inferred latch* to retain the value of Q between process invocations. In general, a VHDL compiler infers a latch for a signal that is assigned a value in an if or case statement if not all input combinations are accounted for.

event attribute

In order to describe edge-triggered behavior of flip-flops, we need to use one of VHDL's predefined signal attributes, the *event* attribute. If "SIG" is a signal name, then the construction "SIG'event" returns the value true at any delta time when SIG changes from one value to another, and false otherwise.

Using the event attribute, we can model a positive-edge triggered flip-flop as shown in Table 8-6. In the IF statement, "CLK'event" returns true on any clock edge, and "CLK='1'" ensures that D is assigned to Q only on a *rising* edge. Note that the process sensitivity list includes only CLK; changes on D at other times are not relevant in this functional model.

BUFFS ‘N’ STUFF

Note that in Table 8-3 we defined the type of Q and QN to be buffer rather than out, since these signals are used as inputs as well as outputs in the architecture definition. Then we had to define a special 2-input NAND gate nand2b with output type buffer, to avoid having a type mismatch (out vs. buffer) in the component instantiations for U4 and U5. Alternatively, we could have used internal signals to get around the problem as shown in Table 8-5. As you know by now, VHDL has many different ways to express the same thing.

Table 8-5 Alternative VHDL structural program for the D latch in Figure 7-12.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdlatch is
  port (D, C: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end Vdlatch;

architecture Vdlatch_s2 of Vdlatch is
  signal DN, SN, RN, IQ, IQN: STD_LOGIC;
  component inv port (I: in STD_LOGIC; O: out STD_LOGIC ); end component;
  component nand2 port (I0, I1: in STD_LOGIC; O: out STD_LOGIC ); end component;
begin
  U1: inv port map (D,DN);
  U2: nand2 port map (D,C,SN);
  U3: nand2 port map (C,DN,RN);
  U4: nand2 port map (SN,IQN,IQ);
  U5: nand2 port map (IQ,RN,IQN);
  Q <= IQ; QN <= IQN;
end Vdlatch_s2;
```

Table 8-6 VHDL behavioral model of an edge-triggered D flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdff is
  port (D, CLK: in STD_LOGIC;
        Q: out STD_LOGIC );
end Vdff;

architecture Vdff_b of Vdff is
begin
process(CLK)
  begin
    if (CLK'event and CLK='1') then Q <= D; end if;
  end process;
end Vdff_b;
```

Table 8-7 VHDL model of a 74x74-like D flip-flop with preset and clear.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vdff74 is
  port (D, CLK, PR_L, CLR_L: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end Vdff74;

architecture Vdff74_b of Vdff74 is
signal PR, CLR: STD_LOGIC;
begin
process(CLR_L, CLR, PR_L, PR, CLK)
  begin
    PR <= not PR_L; CLR <= not CLR_L;
    if (CLR and PR) = '1' then Q <= '0'; QN <= '0';
    elsif CLR = '1' then Q <= '0'; QN <= '1';
    elsif PR = '1' then Q <= '1'; QN <= '0';
    elsif (CLK'event and CLK='1') then Q <= D; QN <= not D;
    end if;
  end process;
end Vdff74_b;
```

Table 8-8 VHDL model of a 16-bit register with many features.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vreg16 is
  port (CLK, CLKEN, OE_L, CLR_L: in STD_LOGIC;
        D: in STD_LOGIC_VECTOR(1 to 16);       -- Input bus
        Q: out STD_ULOGIC_VECTOR (1 to 16) ); -- Output bus (three-state)
end Vreg16;

architecture Vreg16 of Vreg16 is
signal CLR, OE: STD_LOGIC;  -- active-high versions of signals
signal IQ: STD_LOGIC_VECTOR(1 to 16); -- internal Q signals
begin
process(CLK, CLR_L, CLR, OE_L, OE, IQ)
  begin
    CLR <= not CLR_L;  OE <= not OE_L;
    if (CLR = '1') then IQ <= (others => '0');
    elsif (CLK'event and CLK='1') then
      if (CLKEN='1') then IQ <= D; end if;
    end if;
    if OE = '1' then Q <= To_StdULogicVector(IQ);
    else Q <= (others => 'Z'); end if;
  end process;
end Vreg16;
```

The D-flip-flop model can be augmented to include asynchronous inputs and a complemented output as in the 74x74 discrete flip-flop, as shown in Table 8-7. This more detailed functional model shows the non-complementary behavior of the `Q` and `QN` outputs when preset and clear are asserted simultaneously. However, it does not include timing behavior such as propagation delay and setup and hold times, which are beyond the scope of the VHDL coverage in this book.

Larger registers can of course be modeled by defining the data inputs and outputs to be vectors, and additional functions can also be included. For example, Table 8-8 models a 16-bit register with three-state outputs and clock-enable, output-enable, and clear inputs. An internal signal vector `IQ` is used to hold the flip-flop outputs, and three-state outputs are defined and enabled as in Section 5.6.4.

SYNTHESIS RESTRICTIONS

In Table 8-8, the first `elsif` statement theoretically could have included all of the conditions needed to assign `D` to `IQ`. That is, it could have read “`elsif (CLK'event) and (CLK='1') and (CLKEN='1') then ...`” instead of using a nested `if` statement to check `CLKEN`. However, it was written as shown for a very pragmatic reason.

Only a subset of the VHDL language can be synthesized by the VHDL compiler that was used to prepare this chapter; this is true of any VHDL compiler today. In particular, use of the “`event`” attribute is limited to the form shown in the example, and a few others, for detecting simple edge-triggered behavior. This gets mapped into edge-triggered D flip-flops during synthesis. The nested `IF` statement that checks `CLKEN` in the example leads to the synthesis of multiplexer logic on the D inputs of these flip-flops.

8.3 Sequential PLDs

8.3.1 Bipolar Sequential PLDs

PAL16R8

The *PAL16R8*, shown in Figure 8-17, is representative of the first generation of sequential PLDs, which used bipolar (TTL) technology. This device has eight primary inputs, eight outputs, and common clock and output-enable inputs, and fits in a 20-pin package.

The PAL16R8’s AND-OR array is exactly the same as the one found in the PAL16L8 combinational PLD. However, the PAL16R8 has edge-triggered D flip-flops between the AND-OR array and its eight outputs, O1–O8. All of the flip-flops are connected to a common clock input, CLK, and change state on the rising edge of the clock. Each flip-flop drives an output pin through a 3-state buffer; the buffers have a common output-enable signal, OE_L. Notice that, like the combinational output pins of a PAL16L8, the registered output pins of the PAL16R8 contain the complement of the signal produced by the AND-OR array.

The possible inputs to the PAL16R8's AND-OR array are eight primary inputs (I1–I8) and the eight D flip-flop outputs. The connection from the D flip-flop outputs into the AND-OR array makes it easy to design shift registers, counters, and general state machines. Unlike the PAL16L8's combinational outputs, the PAL16R8's D-flip-flop outputs are available to the AND-OR array whether or not the O1–O8 three-state drivers are enabled. Thus, the internal flip-flops can go to a next state that is a function of the current state even when the O1–O8 outputs are disabled.

PAL16R6

Many applications require combinational as well as sequential PLD outputs. The manufacturers of bipolar PLDs addressed this need by providing a few variants of the PAL16R8 that omitted the D flip-flops on some output pins, and instead provided input and output capability identical to that of the PAL16L8's bidirectional pins. For example, Figure 8-18 is the logic diagram of the *PAL16R6*, which has only six registered outputs. Two pins, IO1 and IO8, are bidirectional, serving both as inputs and as combinational outputs with separate 3-state enables, just like the PAL16L8's bidirectional pins. Thus, the possible inputs to the PAL16R6's AND-OR array are the eight primary inputs (I1–I8), the six D-flip-flop outputs, and the two bidirectional pins (IO1, IO8).

PAL16R4
PAL16R8
PAL20L8
PAL20R4

Table 8-9 shows eight standard bipolar PLDs with differing numbers and types of inputs and outputs. All of the PAL16xx parts in the table use the same AND-OR array, where each output has eight AND gates, each with 16 variables and their complements as possible inputs. The PAL20xx parts use a similar

Table 8-9 Characteristics of standard bipolar PLDs.

			Inputs to AND array			
Part number	**Package pins**	**AND-gate inputs**	**Primary inputs**	**Bidirectional combinational outputs**	**Registered outputs**	**Combinational outputs**
PAL16L8	20	16	10	6	0	2
PAL16R4	20	16	8	4	4	0
PAL16R6	20	16	8	2	6	0
PAL16R8	20	16	8	0	8	0
PAL20L8	24	20	14	6	0	2
PAL20R4	24	20	12	4	4	0
PAL20R6	24	20	12	2	6	0
PAL20R8	24	20	12	0	8	0

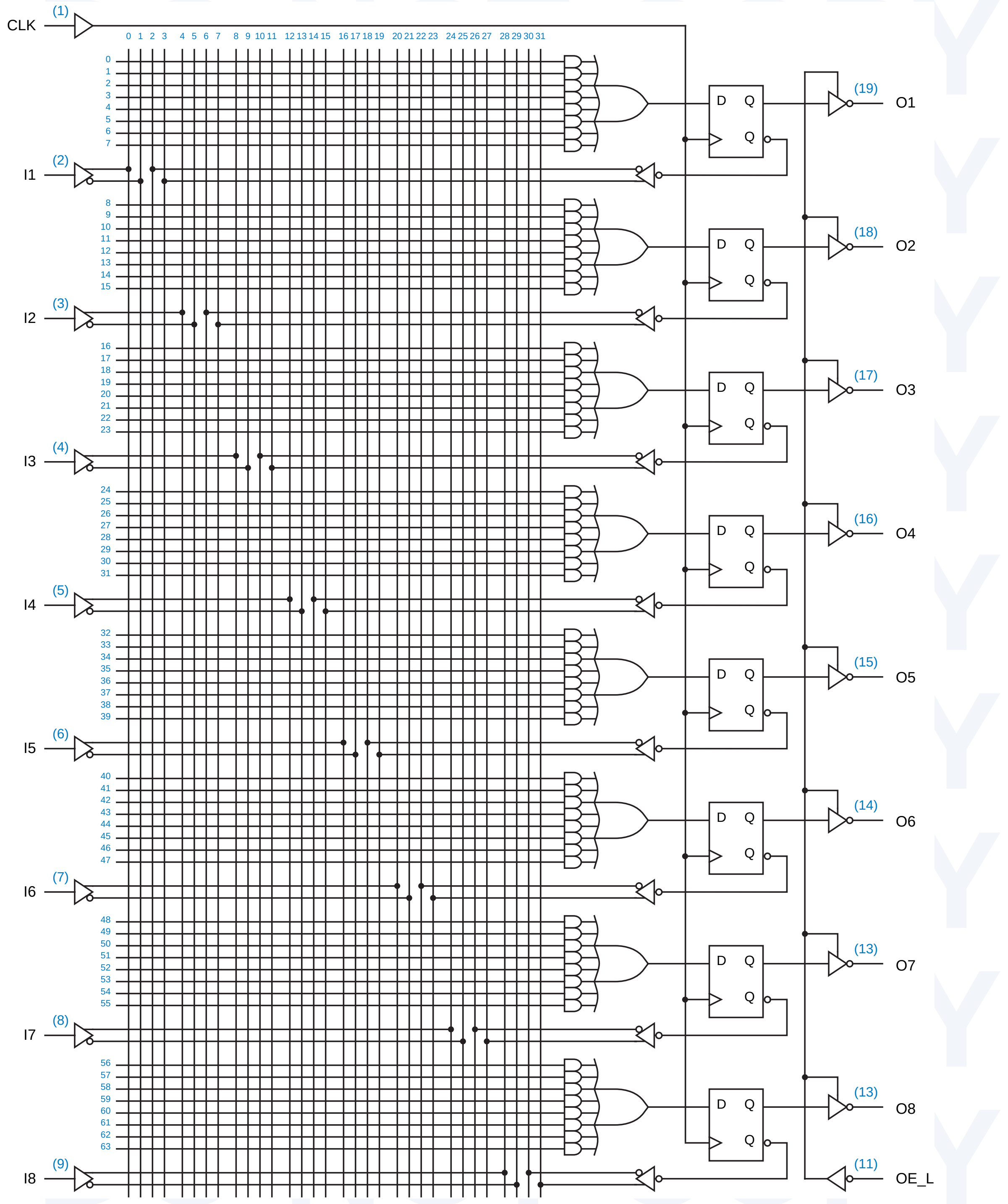


Figure 8-17 PAL16R8 logic diagram.

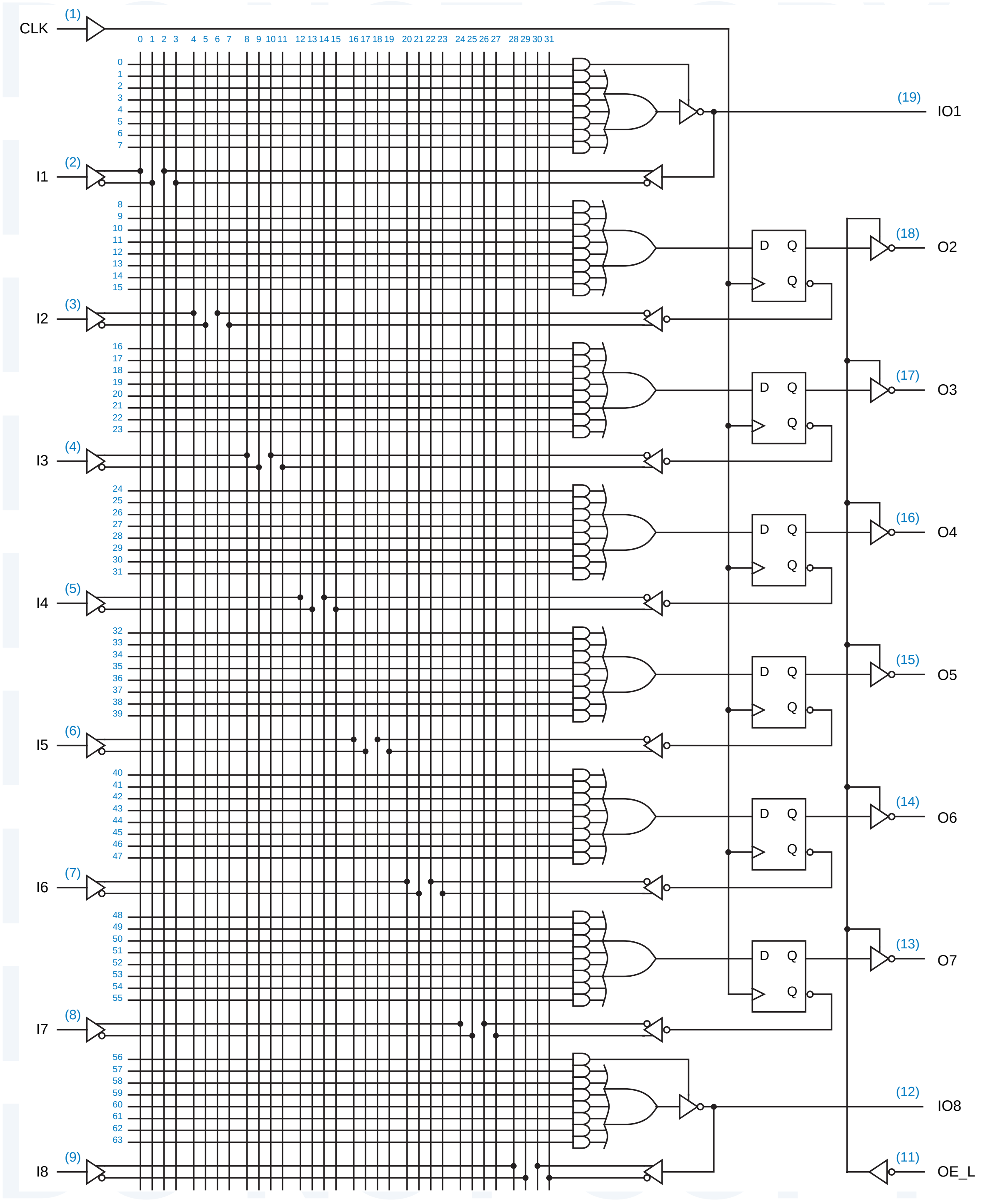


Figure 8-18 PAL16R6 logic diagram.

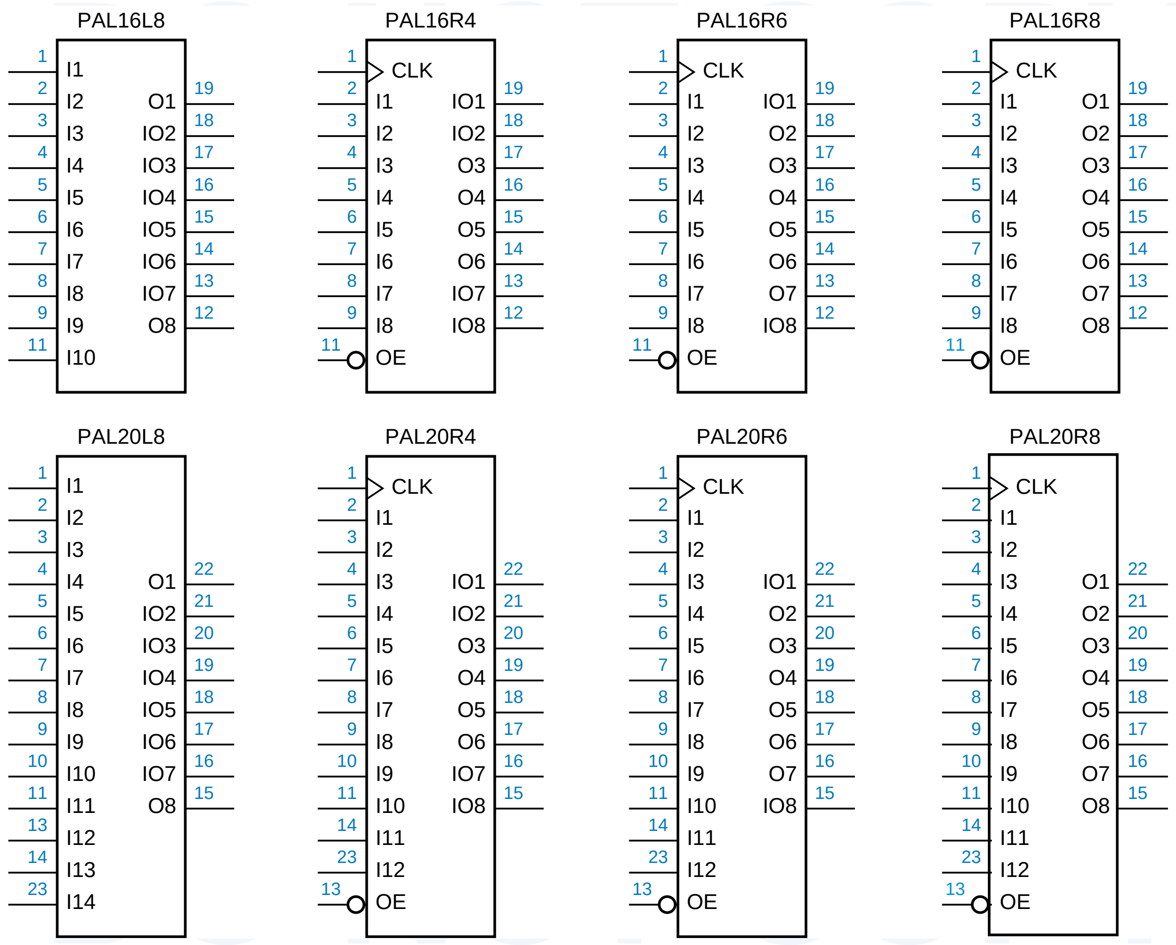


Figure 8-19 Logic symbols for bipolar combinational and sequential PLDs.

AND-OR array with 20 variables and their complements as possible inputs. Figure 8-19 shows logic symbols for all of the PLDs in the table.

PAL20R6
PAL20R8

8.3.2 Sequential GAL Devices

The GAL16V8 electrically erasable PLD was introduced in Section 5.3.3. Two “architecture-control” fuses are used to select among three basic configurations of this device. Section 5.3.3 described the *16V8C* (“complex”) configuration, shown in Figure 5-27 on page 307, a structure similar to that of a bipolar combinational PAL device, the PAL16L8. The *16V8S* (“simple”) configuration provides a slightly different combinational logic capability (see box on page 589).

16V8C
16V8S

The third configuration, called the *16V8R*, allows a flip-flop to be provided on any or all of the outputs. Figure 8-20 shows the structure of the device when

16V8R

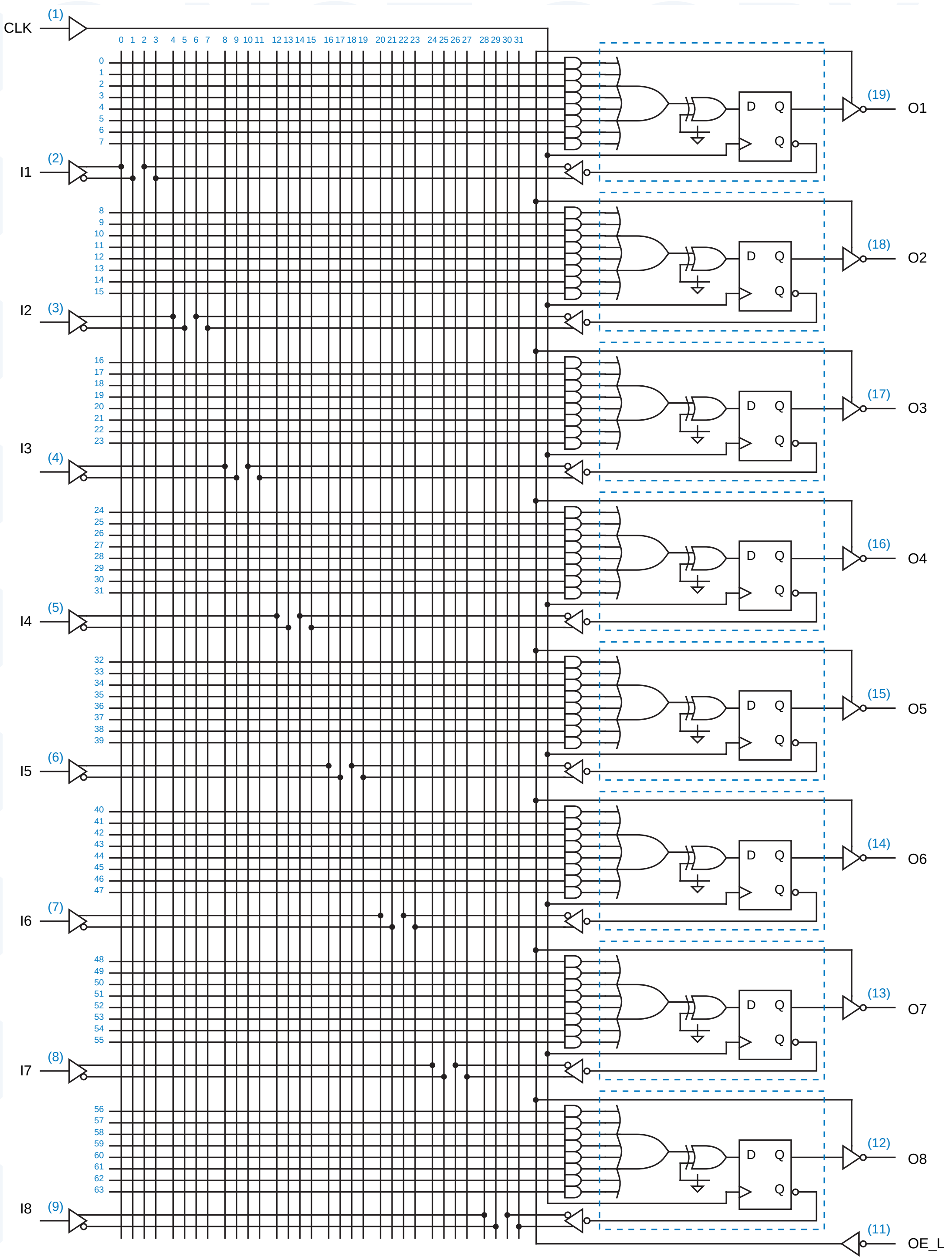
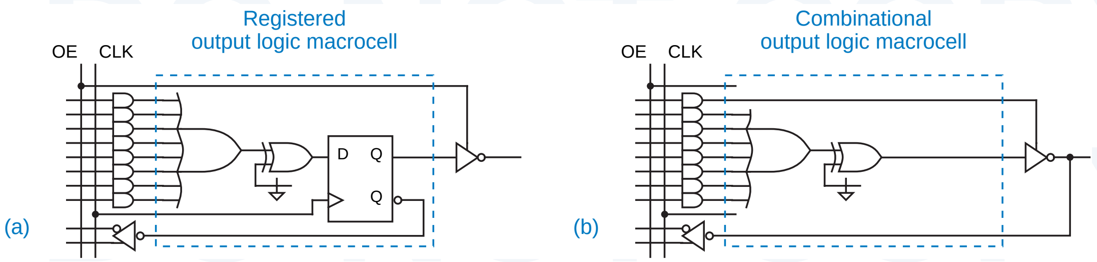


Figure 8-20 Logic diagram for the 16V8 in the “registered” configuration.



(a)
OE
CLK
Registered output logic macrocell
D
Q
Q
(b)
OE
CLK
Combinational output logic macrocell

Figure 8-21 Output logic macrocells for the 16V8R: (a) registered; (b) combinational.

flip-flops are provided on all outputs. Notice that all of the flip-flops are controlled by a common clock signal on pin 1, as in the bipolar devices of the preceding subsection. Likewise, all of the output buffers are controlled by a common output-enable signal on pin 11.

output logic macrocell

The circuitry inside each dotted box in Figure 8-20 is called an *output logic macrocell*. The 16V8R is much more flexible than a PAL16R8 because each macrocell may be individually configured to bypass the flip-flop, that is, to produce a combinational output. Figure 8-21 shows the two macrocell configurations that are possible in the 16V8R; (a) is registered and (b) is combinational. Thus, it is possible to program the device to have any set of registered and combinational outputs, up to eight total.

20V8

The *20V8* is similar to the 16V8, but comes in a 24-pin package with four extra input-only pins. Each AND gate in the 20V8 has 20 inputs, hence the “20” in “20V8.”

THE “SIMPLE” 16V8S

The “simple” 16V8S configuration of the GAL16V8 is not often used, because its capabilities are mostly a subset of the 16V8C’s. Instead of an AND term, the 16V8S uses one fuse per output to control whether the output buffers are enabled. That is, each output pin may be programmed either to be always enabled or to be always disabled (except pins 15 and 16, which are always enabled). All of the output pins (except 15 and 16) are available as inputs to the AND array regardless of whether the output buffer is enabled.

The only advantage of a 16V8S compared to a 16V8C is that it has eight, not seven, AND terms as inputs to the OR gate on each output. The 16V8S architecture was designed mainly for emulation of certain now-obsolete bipolar PAL devices, some of which either had eight product terms per output or had inputs on pins 12 and 19, which are not inputs in the 16V8C configuration. With appropriate programming, the 16V8S can be used as a pin-for-pin compatible replacement for these devices, which included the PAL10H8, PAL12H6, PAL14H4, PAL16H2, PAL10L8, PAL12L6, PAL14L4, and PAL16L2.

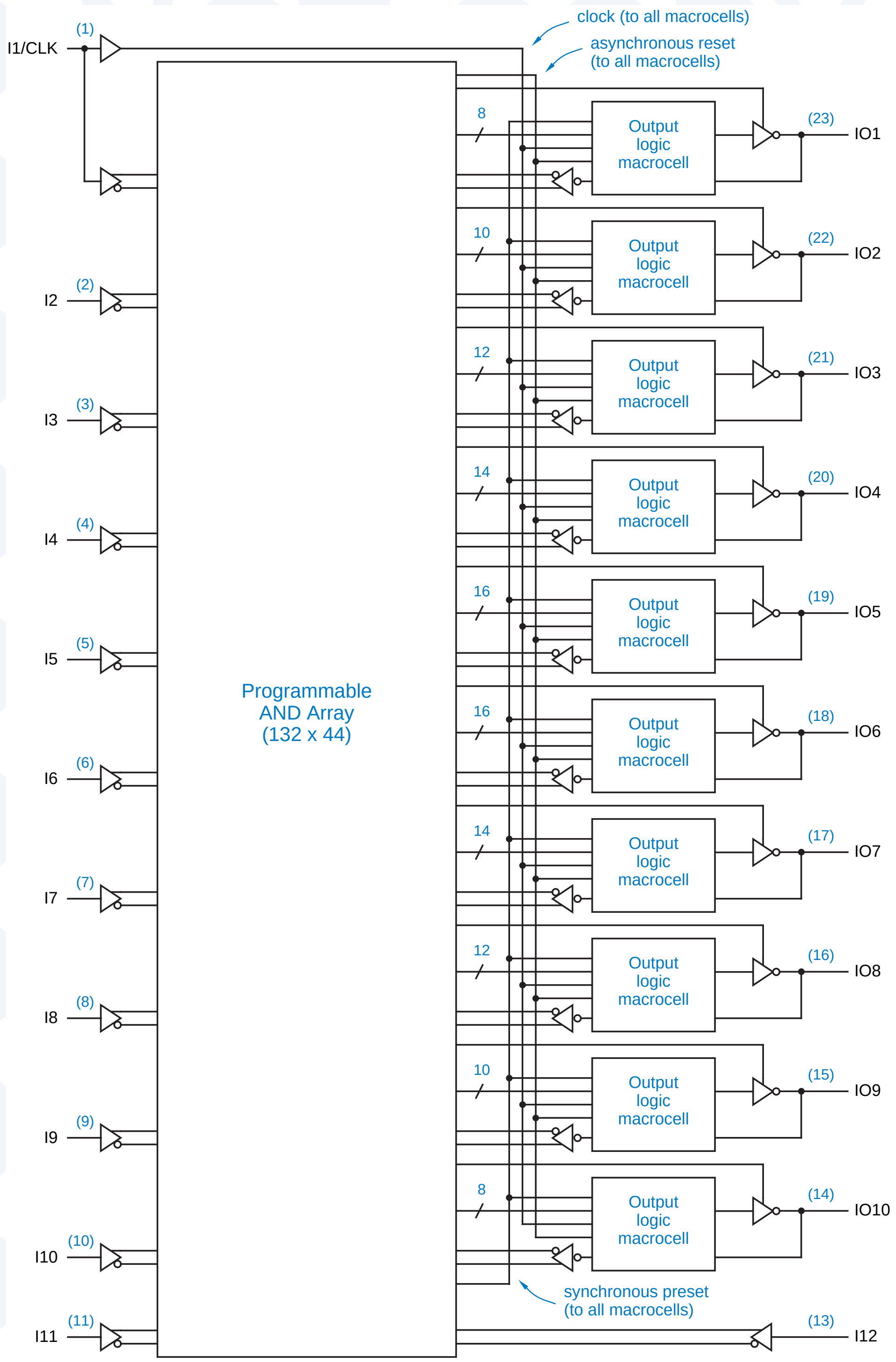


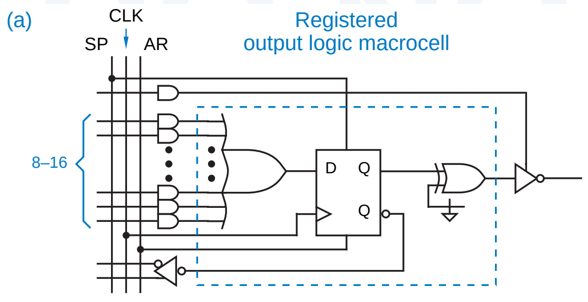
Figure 8-22
Logic diagram for the 22V10.

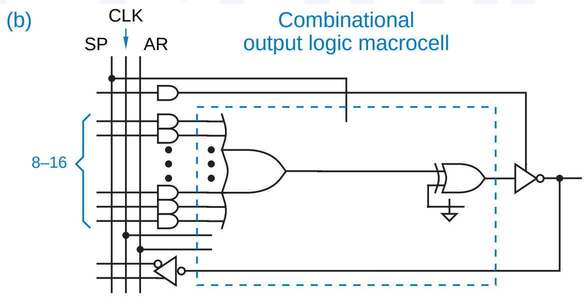
22V10

The *22V10*, whose basic structure is shown in Figure 8-22, also comes in a 24-pin package, but is somewhat more flexible than the 20V8. The 22V10 does not have "architecture control" bits like the 16V8's and 20V8's, but it can realize any function that is realizable with a 20V8, and more. It has more product terms, two more general-purpose inputs, and better output-enable control than the 20V8. Key differences are summarized below:

- Each output logic macrocell is configurable to have a register or not, as in the 20V8R architecture. However, the macrocells are different from the 16V8's and 20V8's, as shown in Figure 8-23.
- A single product term controls the output buffer, regardless of whether the registered or the combinational configuration is selected for a macrocell.
- Every output has at least eight product terms available, regardless of whether the registered or the combinational configuration is selected. Even more product terms are available on the inner pins, with 16 available on each of the two innermost pins. ("Innermost" is with respect to the right-hand side of the Figure 8-22, which also matches the arrangement of these pins on a 24-pin dual-inline package.)
- The clock signal on pin 1 is also available as a combinational input to any product term.
- A single product term is available to generate a global, asynchronous reset signal that resets all internal flip-flops to 0.
- A single product term is available to generate a global, synchronous preset signal that sets all internal flip-flops to 1 on the rising edge of the clock.
- Like the 16V8 and 20V8, the 22V10 has programmable output polarity. However, in the registered configuration, the polarity change is made at the output, rather than the input, of the D flip-flop. This affects the details of programming when the polarity is changed but does not affect the overall capability of the 22V10 (i.e., whether a given function can be realized). In fact, the difference in polarity-change location is transparent when you use a PLD programming language such as ABEL.

Figure 8-23 Output logic macrocells for the 22V10: (a) registered; (b) combinational.





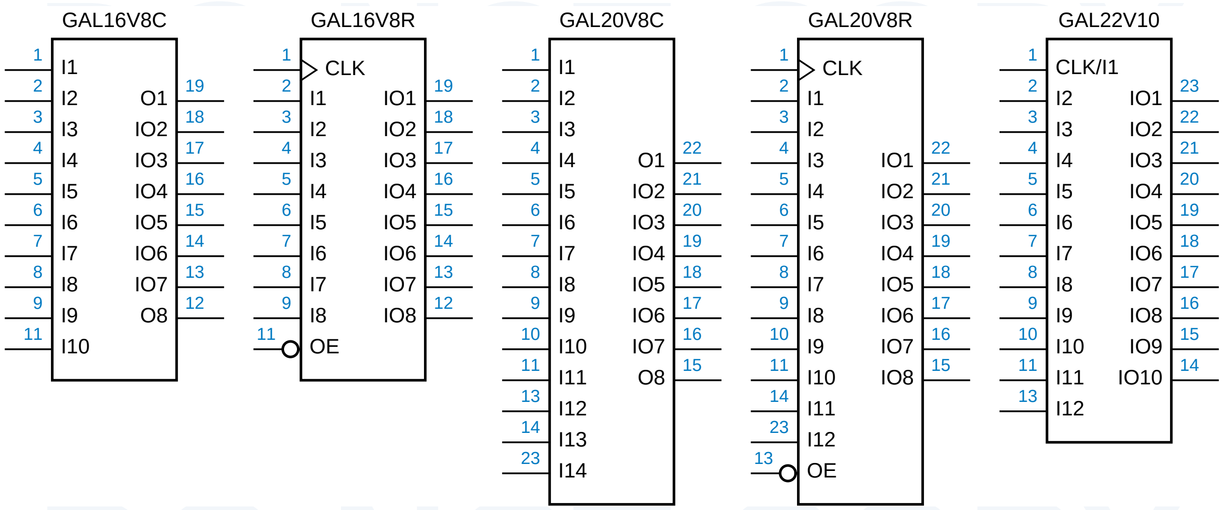


Figure 8-24 Logic symbols for popular GAL devices.

For most of the 1990s, the 16V8, 20V8, and 22V10 were the most popular and cost-effective PLDs (but see the box on page 595). Figure 8-24 shows generic logic symbols for these three devices. Most of the examples in the rest of this chapter can fit into the smallest of the three devices, the 16V8.

PALS? GALS? Lattice Semiconductor introduced GAL devices including the GAL16V8 and GAL20V8 in the mid-1980s. Advanced Micro Devices later followed up with a pin-compatible device which they call the PALCE16V8 (“C” is for CMOS, “E” is for erasable). Several other manufacturers make differently numbered but compatible devices as well. Rather than get caught up in the details of different manufacturers’ names, in this chapter we usually refer to commonly used GAL devices with their generic names, 16V8, 20V8, and 22V10.

8.3.3 PLD Timing Specifications

Several timing parameters are specified for combinational and sequential PLDs. The most important ones are illustrated in Figure 8-25 and are explained below:

t_{PD}

feedback input

t_{PD} This parameter applies to combinational outputs. It is the propagation delay from a primary input pin, bidirectional pin, or “feedback” input to the combinational output. A *feedback input* is an internal input of the AND-OR array that is driven by the registered output of an internal macrocell.

t_{CO} This parameter applies to registered outputs. It is the propagation delay from the rising edge of CLK to a primary output. t_{CO}

t_{CF} This parameter also applies to registered outputs. It is the propagation delay from the rising edge of CLK to a macrocell's registered output that connects back to a feedback input. If specified, t_{CF} is normally less than t_{CO}. However, some manufacturers do not specify t_{CF}, in which case you must assume that $t_{CF} = t_{CO}$. t_{CF}

t_{SU} This parameter applies to primary, bidirectional, and feedback inputs that propagate to the D inputs of flip-flops. It is the setup time that the input signal must be stable before the rising edge of CLK. t_{SU}

t_H This parameter also applies to signals that propagate to the D inputs of flip-flops. It is the hold time that the input signal must be stable after the rising edge of CLK. t_H

f_{max} This parameter applies to clocked operation. It is the highest frequency at which the PLD can operate reliably, and is the reciprocal of the minimum clock period. Two versions of this parameter can be derived from the previous specifications, depending on whether the device is operating with external feedback or internal feedback. f_{max}

External feedback refers to a circuit in which a registered PLD output is connected to the input of another registered PLD with similar timing; for proper operation, the sum of t_{CO} for the first PLD and t_{SU} for the second must not exceed the clock period. *external feedback*

Internal feedback refers to a circuit in which a registered PLD output is fed back to a register in the same PLD; in this case, the sum of t_{CF} and t_{SU} must not exceed the clock period. *internal feedback*

Each of the PLDs that we described in previous sections is available in several different speed grades. The speed grade is usually indicated by a suffix on the part number, such as “16V8-10”; the suffix usually refers to the t_{PD}

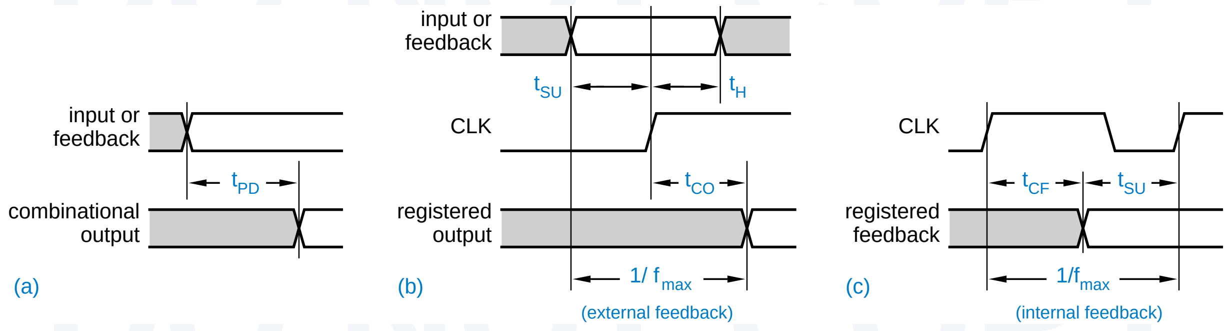


Figure 8-25 PLD timing parameters.

Table 8-10 Timing specifications, in nanoseconds, of popular bipolar and CMOS PLDs.

Part numbers	Suffix	t_{PD}	t_{CO}	t_{CF}	t_{SU}	t_H
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	-5	5	4	–	4.5	0
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	-7	7.5	6.5	–	7	0
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	-10	10	8	–	10	0
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	B	15	12	–	15	0
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	B-2	25	15	–	25	0
PAL16L8, PAL16Rx, PAL20L8, PAL20Rx	A	25	15	–	25	0
PALCE16V8, PALCE20V8	-5	5	4	–	3	0
GAL16V8, GAL20V8	-7	7.5	5	3	5	0
GAL16V8, GAL20V8	-10	10	7.5	6	7.5	0
GAL16V8, GAL20V8	-15	15	10	8	12	0
GAL16V8, GAL20V8	-25	25	12	10	15	0
PALCE22V10	-5	5	4	–	3	0
PALCE22V10	-7	7.5	4.5	–	4.5	0
GAL22V10	-10	10	7	2.5	7	0
GAL22V10	-15	15	8	2.5	10	0
GAL22V10	-25	25	15	13	15	0

specification, in nanoseconds. Table 8-10 shows the timing of several popular bipolar and CMOS PLDs. Note that only the t_{PD} column applies to the combinational outputs of a device, while the last four columns apply to registered outputs. All of the timing specifications are worst-case numbers over the commercial operating range.

When sequential PLDs are used in applications with critical timing, it's important to remember that they normally have longer setup times than discrete edge-triggered registers in the same technology, owing to the delay of the AND-OR array on each D input. Conversely, under typical conditions, a PLD actually has a negative hold-time requirement because of the delay through AND-OR array. However, you can't count on it having a negative hold time—the worst-case specification is normally zero.

HOW MUCH DOES IT COST?

Once you understand the capabilities of different PLDs, you might ask, “Why not just always use the most capable PLD available?” For example, even if a circuit fits in a 20-pin 16V8, why not specify the slightly larger, 24-pin 20V8 so that spare inputs are available in case of trouble? And, once you’ve specified a 20V8, why not use the somewhat more capable 22V10 which comes in the same 24-pin package?

In the real world of product design and engineering, the constraint is cost. Otherwise, the argument of the previous paragraph could be extended *ad nauseum*, using CPLDs and FPGAs with even more capability (see \chapref{CPLDsFPGAs}).

Like automobiles and fine wines, digital devices such as PLDs, CPLDs, and FPGAs are not always priced proportionally to their capabilities and benefits. In particular, the closer a device’s capability is to the “bleeding edge,” the higher the premium you can expect to pay. Thus, when selecting a devices to realize a design, you must evaluate many trade-offs. For example, a high-density, high-cost CPLD or FPGA may allow a design to be realized in a single device whose internal functions are easily changed if need be. On the other hand, using two or more lower density PLDs, CPLDs, or FPGAs may save component cost but increase board area and power consumption, while making it harder to change the design later (since the device interconnections must be fixed when the board is fabricated).

What this goes to show is that overall cost must always be considered along with design elegance and convenience to create a successful (i.e., profitable) product. And minimizing the cost of a product usually involves a plethora of common-sense economic and engineering considerations that are far removed from the turn-the-crank, algorithmic gate minimization methods of Chapter 4.

8.4 Counters

The name *counter* is generally used for any clocked sequential circuit whose state diagram contains a single cycle, as in Figure 8-26. The *modulus* of a counter is the number of states in the cycle. A counter with *m* states is called a *modulo-m counter* or, sometimes, a *divide-by-m counter.* A counter with a non-power-of-2 modulus has extra states that are not used in normal operation.

counter
modulus
modulo-m counter
divide-by-m counter

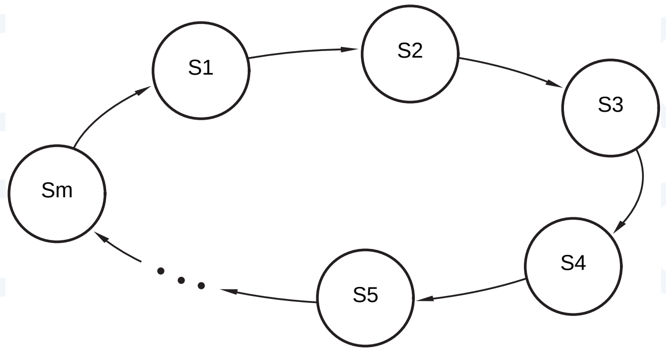


Figure 8-26 General structure of a counter’s state diagram—a single cycle.

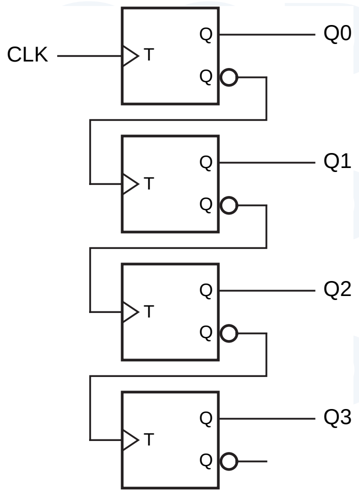


Figure 8-27
A 4-bit binary ripple counter.

n-bit binary counter

Probably the most commonly used counter type is an *n-bit binary counter*. Such a counter has *n* flip-flops and has 2^n states, which are visited in the sequence 0, 1, 2, … , 2^n–1, 0, 1, … . Each of these states is encoded as the corresponding *n*-bit binary integer.

8.4.1 Ripple Counters

An *n*-bit binary counter can be constructed with just *n* flip-flops and no other components, for any value of *n*. Figure 8-27 shows such a counter for $n = 4$. Recall that a T flip-flop changes state (toggles) on every rising edge of its clock input. Thus, each bit of the counter toggles if and only if the immediately preceding bit changes from 1 to 0. This corresponds to a normal binary counting sequence—when a particular bit changes from 1 to 0, it generates a carry to the next most significant bit. The counter is called a *ripple counter* because the carry information ripples from the less significant bits to the more significant bits, one bit at a time.

ripple counter

8.4.2 Synchronous Counters

Although a ripple counter requires fewer components than any other type of binary counter, it does so at a price—it is slower than any other type of binary counter. In the worst case, when the most significant bit must change, the output is not valid until time $n \cdot t_{TQ}$ after the rising edge of CLK, where t_{TQ} is the propagation delay from input to output of a T flip-flop.

synchronous counter

A *synchronous counter* connects all of its flip-flop clock inputs to the same common CLK signal, so that all of the flip-flop outputs change at the same time, after only t_{TQ} ns of delay. As shown in Figure 8-28, this requires the use of T flip-flops with enable inputs; the output toggles on the rising edge of T if and only if EN is asserted. Combinational logic on the EN inputs determines which, if any, flip-flops toggle on each rising edge of T.

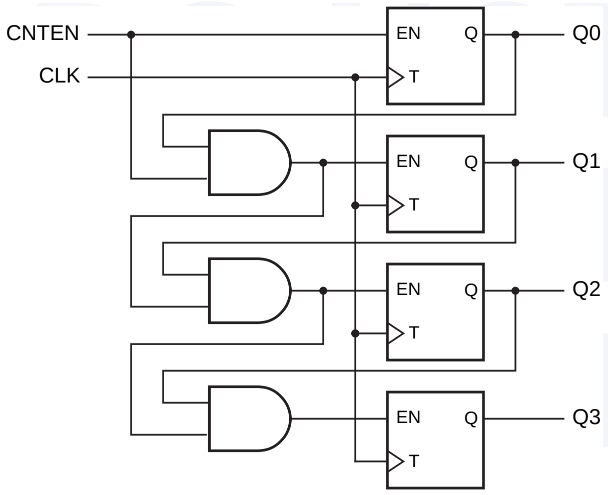


Figure 8-28
A synchronous 4-bit binary counter with serial enable logic.

As shown in Figure 8-28, it is also possible to provide a master count-enable signal CNTEN. Each T flip-flop toggles if and only if CNTEN is asserted and all of the lower-order counter bits are 1. Like the binary ripple counter, a synchronous *n*-bit binary counter can be built with a fixed amount of logic per bit—in this case, a T flip-flop with enable and a 2-input AND gate.

synchronous serial counter

The counter structure in Figure 8-28 is sometimes called a *synchronous serial counter* because the combinational enable signals propagate serially from the least significant to the most significant bits. If the clock period is too short, there may not be enough time for a change in the counter's LSB to propagate to the MSB. This problem is eliminated in Figure 8-29 by driving each EN input with a dedicated AND gate, just a single level of logic. Called a *synchronous parallel counter,* this is the fastest binary counter structure.

synchronous parallel counter

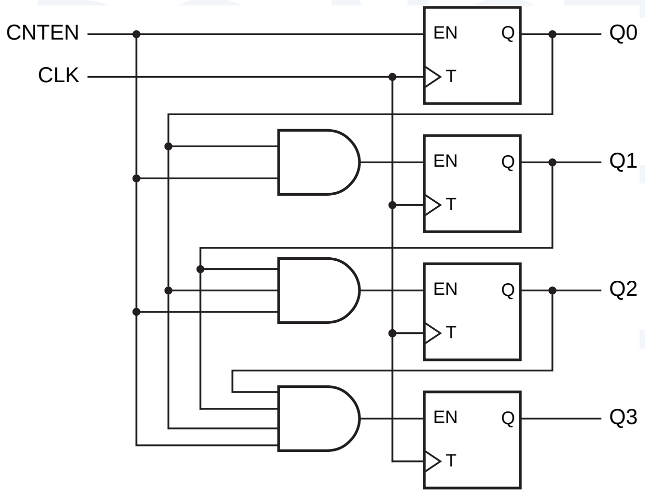


Figure 8-29
A synchronous 4-bit binary counter with parallel enable logic

8.4.3 MSI Counters and Applications

74x163

The most popular MSI counter is the *74x163*, a synchronous 4-bit binary counter with active-low load and clear inputs, with the traditional logic symbol shown in Figure 8-30. Its function is summarized by the state table in Table 8-11, and its internal logic diagram is shown in Figure 8-31.

The '163 uses D flip-flops rather than T flip-flops internally to facilitate the load and clear functions. Each D input is driven by a 2-input multiplexer consisting of an OR gate and two AND gates. The multiplexer output is 0 if the CLR_L input is asserted. Otherwise, the top AND gate passes the data input (A, B, C, or D) to the output if LD_L is asserted. If neither CLR_L nor LD_L is asserted, the bottom AND gate passes the output of an XNOR gate to the multiplexer output.

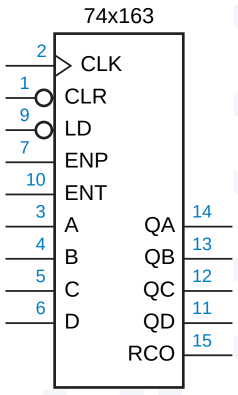


Figure 8-30 Traditional logic symbol for the 74x163.

Table 8-11 State table for a 74x163 4-bit binary counter.

Inputs				Current State				Next State			
CLR_L	***LD_L***	***ENT***	***ENP***	***QD***	***QC***	***QB***	***QA***	***QD****	***QC****	***QB****	***QA****
0	x	x	x	x	x	x	x	0	0	0	0
1	0	x	x	x	x	x	x	D	C	B	A
1	1	0	x	x	x	x	x	QD	QC	QB	QA
1	1	x	0	x	x	x	x	QD	QC	QB	QA
1	1	1	1	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	0	1	1
1	1	1	1	0	0	1	1	0	1	0	0
1	1	1	1	0	1	0	0	0	1	0	1
1	1	1	1	0	1	0	1	0	1	1	0
1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0	1	0	0	1
1	1	1	1	1	0	0	1	1	0	1	0
1	1	1	1	1	0	1	0	1	0	1	1
1	1	1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	0	0	1	1	0	1
1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0

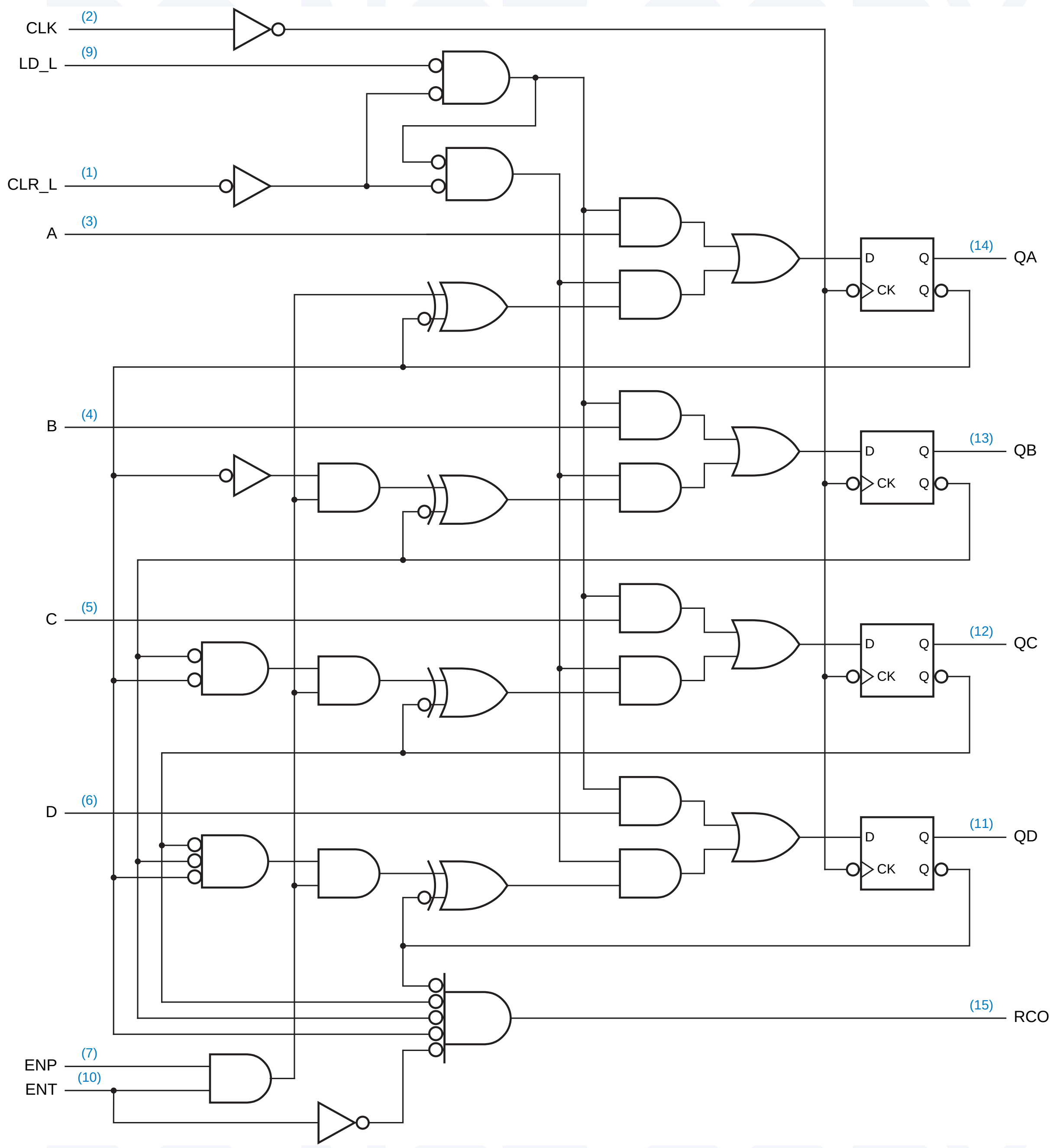


Figure 8-31 Logic diagram for the 74x163 synchronous 4-bit binary counter, including pin numbers for a standard 16-pin dual in-line package.

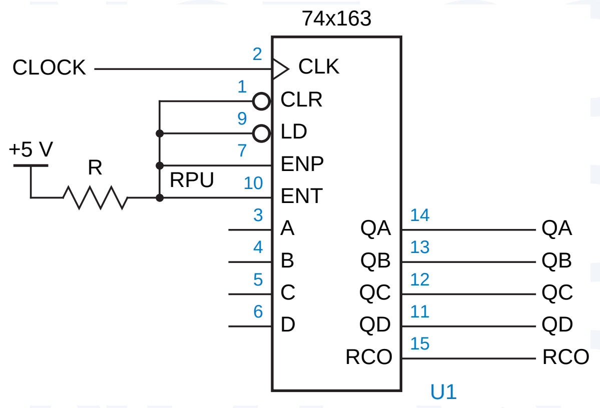


Figure 8-32 Connections for the 74x163 to operate in a free-running mode.

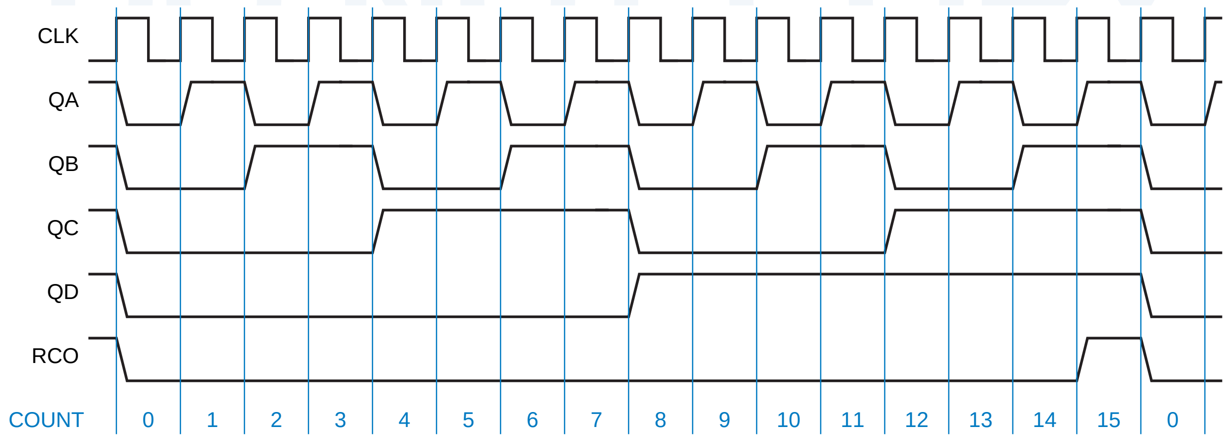
The XNOR gates perform the counting function in the ’163. One input of each XNOR is the corresponding count bit (QA, QB, QC, or QD); the other input is 1, which complements the count bit, if and only if both enables ENP and ENT are asserted and all of the lower-order count bits are 1. The RCO (“ripple carry out”) signal indicates a carry from the most significant bit position, and is 1 when all of the count bits are 1 and ENT is asserted.

free-running counter

Even though most MSI counters have enable inputs, they are often used in a *free-running* mode in which they are enabled continuously. Figure 8-32 shows the connections to make a ’163 operate in this way, and Figure 8-33 shows the resulting output waveforms. Notice that starting with QA, each signal has half the frequency of the preceding one. Thus, a free-running ’163 can be used as a divide-by-2, -4, -8, or -16 counter, by ignoring any unnecessary high-order output bits.

Note that the ’163 is fully synchronous; that is, its outputs change only on the rising edge of CLK. Some applications need an asynchronous clear function,

Figure 8-33 Clock and output waveforms for a free-running divide-by-16 counter.



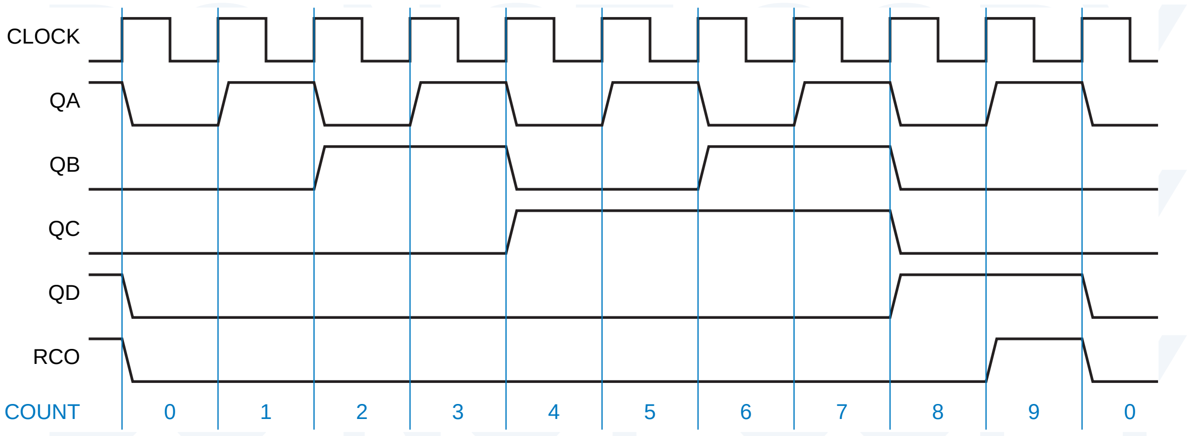


Figure 8-34 Clock and output waveforms for a free-running divide-by-10 counter.

as provided by the *74x161*. The '161 has the same pinout as the '163, but its CLR_L input is connected to the asynchronous clear inputs of its flip-flops. *74x161*

The *74x160* and *74x162* are more variations with the same pinouts and general functions as the '161 and '163, except that the counting sequence is modified to go to state 0 after state 9. In other words, these are modulo-10 counters, sometimes called *decade counters*. Figure 8-34 shows the output waveforms for a free-running '160 or '162. Notice that although the QD and QC outputs have one-tenth of the CLK frequency, they do not have a 50% duty cycle, and the QC output, with one-fifth of the input frequency, does not have a constant duty cycle. We'll show the design of a divide-by-10 counter with a 50% duty-cycle output later in this subsection. *74x160* *74x162* *decade counter*

Although the '163 is a modulo-16 counter, it can be made to count in a modulus less than 16 by using the CLR_L or LD_L input to shorten the normal counting sequence. For example, Figure 8-35 shows one way of using the '163 as a modulo-11 counter. The RCO output, which detects state 15, is used to force

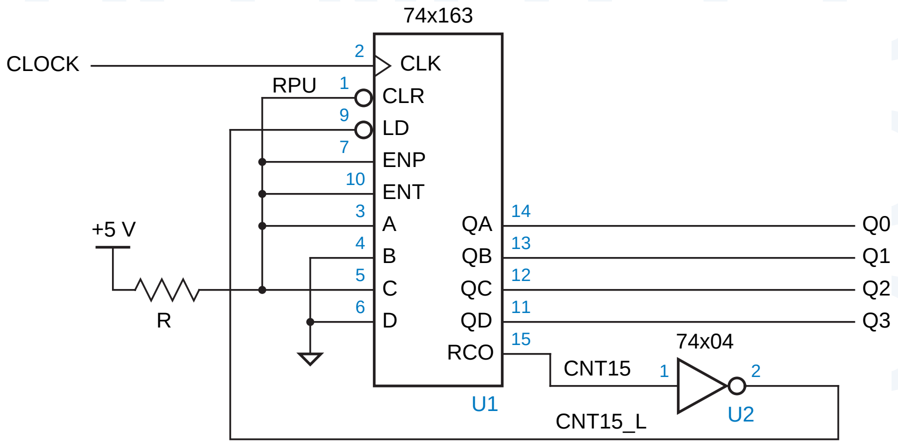


Figure 8-35 Using the 74x163 as a modulo-11 counter with the counting sequence 5, 6, …, 15, 5, 6, ….

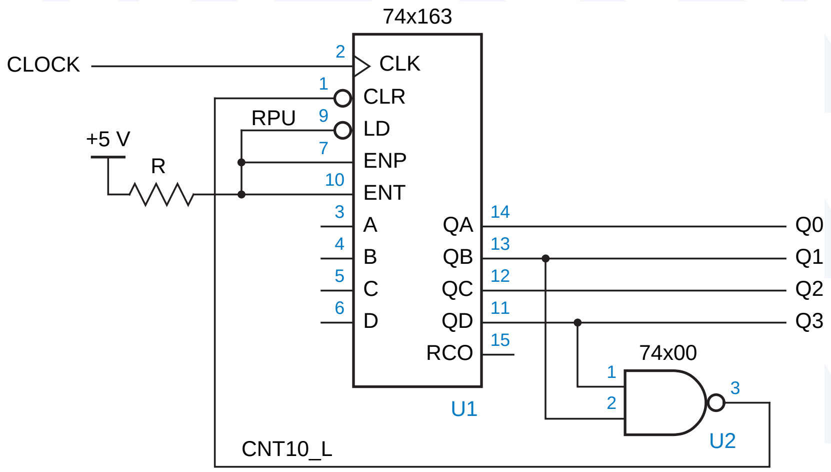


Figure 8-36 Using the 74x163 as a modulo-11 counter with the counting sequence 0, 1, 2, …, 10, 0, 1, ….

the next state to 5, so that the circuit will count from 5 to 15 and then start at 5 again, for a total of 11 states per counting cycle.

A different approach for modulo-11 counting with the ’163 is shown in Figure 8-36. This circuit uses a NAND gate to detect state 10 and force the next state to 0. Notice that only a 2-input gate is used to detect state 10 (binary 1010). Although a 4-input gate would normally be used to detect the condition CNT10 = Q3 · Q2′ · Q1 · Q0′, the 2-input gate takes advantage of the fact that no other state in the normal counting sequence of 0–10 has Q3 = 1 and Q1 = 1. In general, to detect state *N* in a binary counter that counts from 0 to *N*, we need to AND only the state bits that are 1 in the binary encoding of *N*.

There are many other ways to make a modulo-11 counter using a ’163. The choice of approach—one of the preceding or a combination of them (as in Exercise 8.25)—depends on the application. As another example, in Section 2.10 we promised to show you how to build a circuit that counts in the

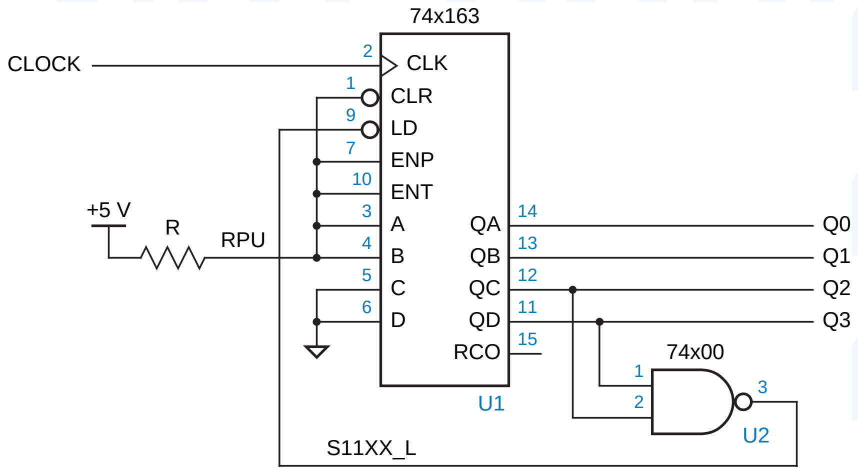


Figure 8-37 A 74x163 used as an excess-3 decimal counter.

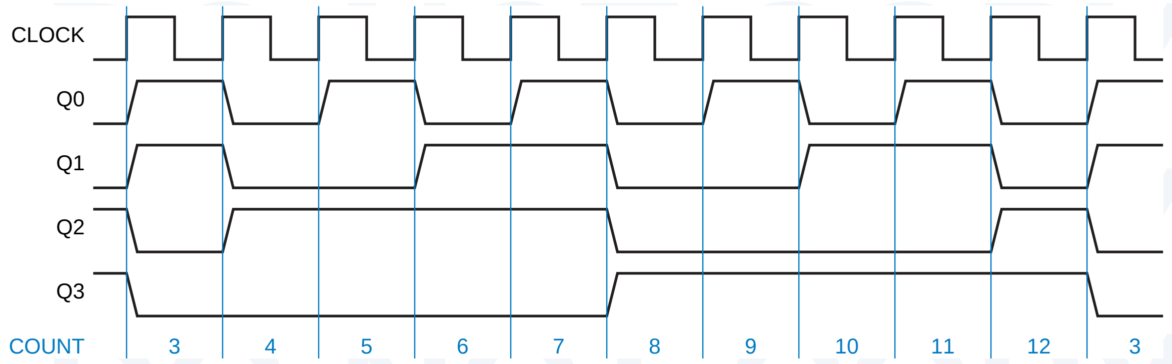
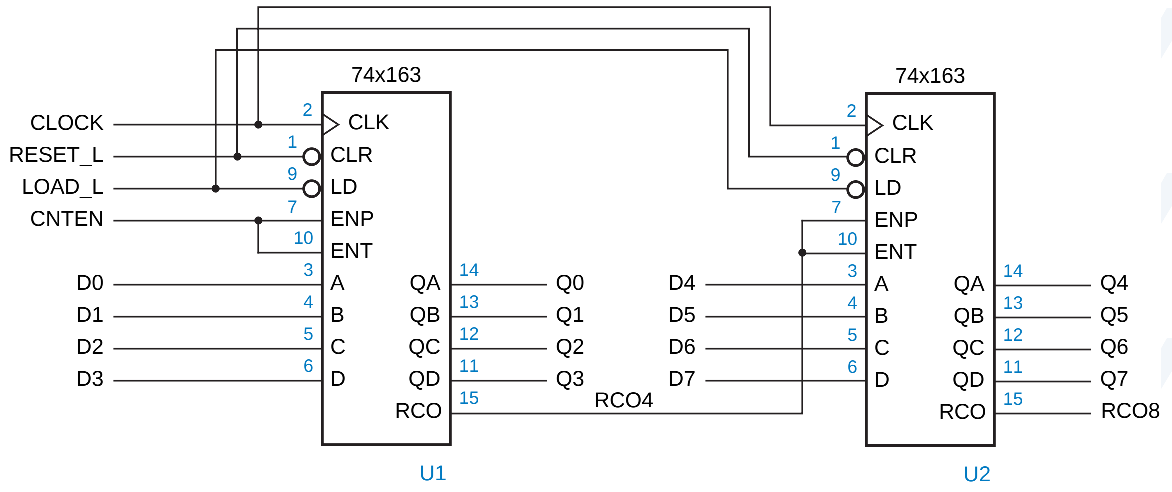


Figure 8-38 Timing waveforms for the '163 used as an excess-3 decimal counter.

excess-3 decimal code, shown in Table 2-9 on page 45. Figure 8-37 shows the connections for a '163 to count in the excess-3 sequence. A NAND gate detects state 1100 and forces 0011 to be loaded as the next state. Figure 8-38 shows the resulting timing waveforms. Notice that the Q3 output has a 50% duty cycle, which may be desirable for some applications.

A binary counter with a modulus greater than 16 can be built by cascading 74x163s. Figure 8-39 shows the general connections for such a counter. The CLK, CLR_L, and LD_L inputs of all the '163s are connected in parallel, so that all of them count or are cleared or loaded at the same time. A master count-enable (CNTEN) signal is connected to the low-order '163. The RCO4 output is asserted if and only if the low-order '163 is in state 15 *and* CNTEN is asserted; RCO4 is connected to the enable inputs of the high-order '163. Thus, both the carry information and the master count-enable ripple from the output of one

Figure 8-39 General cascading connections for 74x163-based counters.

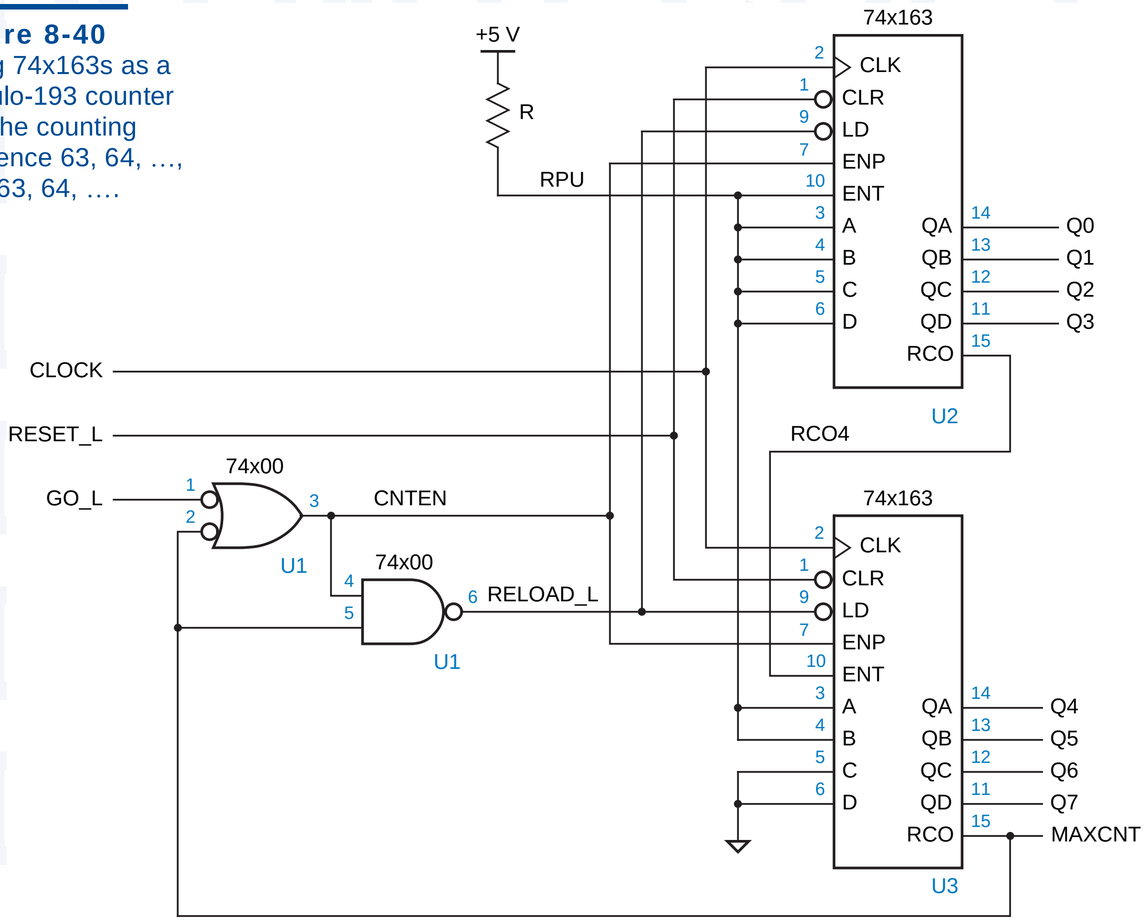


4-bit counter stage to the next. Like the synchronous serial counter of Figure 8-28, this scheme can be extended to build a counter with any desired number of bits; the maximum counting speed is limited by the propagation delay of the ripple carry signal through all of the stages (but see Exercise 8.27).

Even experienced digital designers are sometimes confused about the difference between the ENP and ENT enable inputs of the '163 and similar counters, since both must be asserted for the counter to count. However, a glance at the 163's internal logic diagram, Figure 8-31 on page 599, shows the difference quite clearly—ENT goes to the ripple carry output as well. In many applications, this distinction is important.

For example, Figure 8-40 shows an application that uses two '163s as a modulo-193 counter that counts from 63 to 255. The MAXCNT output detects state 255 and stops the counter until GO_L is asserted. When GO_L is asserted, the counter is reloaded with 63 and counts up to 255 again. (Note that the value of GO_L is relevant only when the counter is in state 255.) To keep the counter

Figure 8-40
Using 74x163s as a modulo-193 counter with the counting sequence 63, 64, …, 255, 63, 64, ….



stopped, MAXCNT must be asserted in state 255 even while the counter is stopped. Therefore, the low-order counter's ENT input is always asserted, its RCO output is connected to the high-order ENT input, and MAXCNT detects state 255 even if CNTEN is not asserted (compare with the behavior of RCO8 in Figure 8-39). To enable counting, CNTEN is connected to the ENP inputs in parallel. A NAND gate asserts RELOAD_L to go back to state 63 only if GO_L is asserted and the counter is in state 255.

74x169

Another counter with functions similar to 74x163's is the *74x169,* whose logic symbol is shown in Figure 8-41. One difference in the '169 is that its carry output and enable inputs are active low. More importantly, the '169 is an *up/down counter;* it counts in ascending or descending binary order depending on the value of an input signal, UP/DN. The '169 counts up when UP/DN is 1 and down when UP/DN is 0.

up/down counter

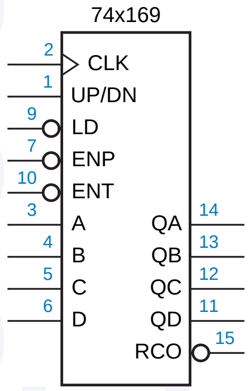


Figure 8-41
Logic symbol for the 74x169 up/down counter.

8.4.4 Decoding Binary-Counter States

A binary counter may be combined with a decoder to obtain a set of 1-out-of-*m*-coded signals, where one signal is asserted in each counter state. This is useful when counters are used to control a set of devices where a different device is enabled in each counter state. In this approach, each output of the decoder enables a different device.

Figure 8-42 shows how a 74x163 wired as a modulo-8 counter can be combined with a 74x138 3-to-8 decoder to provide eight signals, each one representing a counter state. Figure 8-43 shows typical timing for this circuit. Each decoder output is asserted during a corresponding clock period.

decoding glitches

Notice that the decoder outputs may contain "glitches" on state transitions where two or more counter bits change, even though the '163 outputs are glitch free and the '138 does not have any static hazards. In a synchronous counter like the '163, the outputs don't change at exactly the same time. More important,

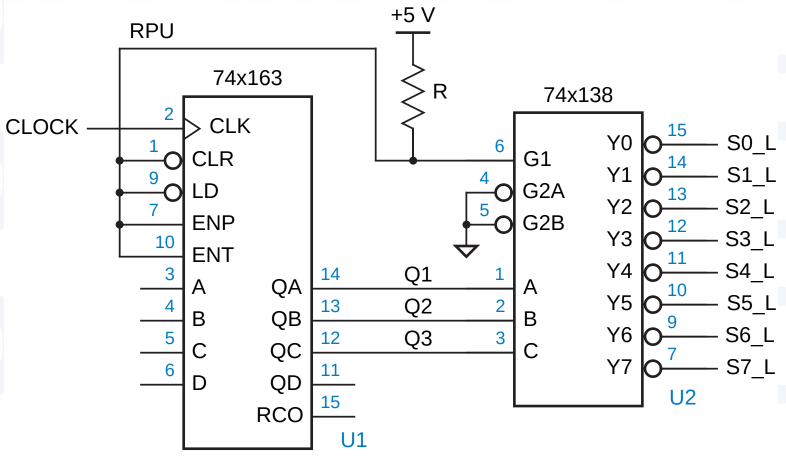


Figure 8-42
A modulo-8 binary counter and decoder.

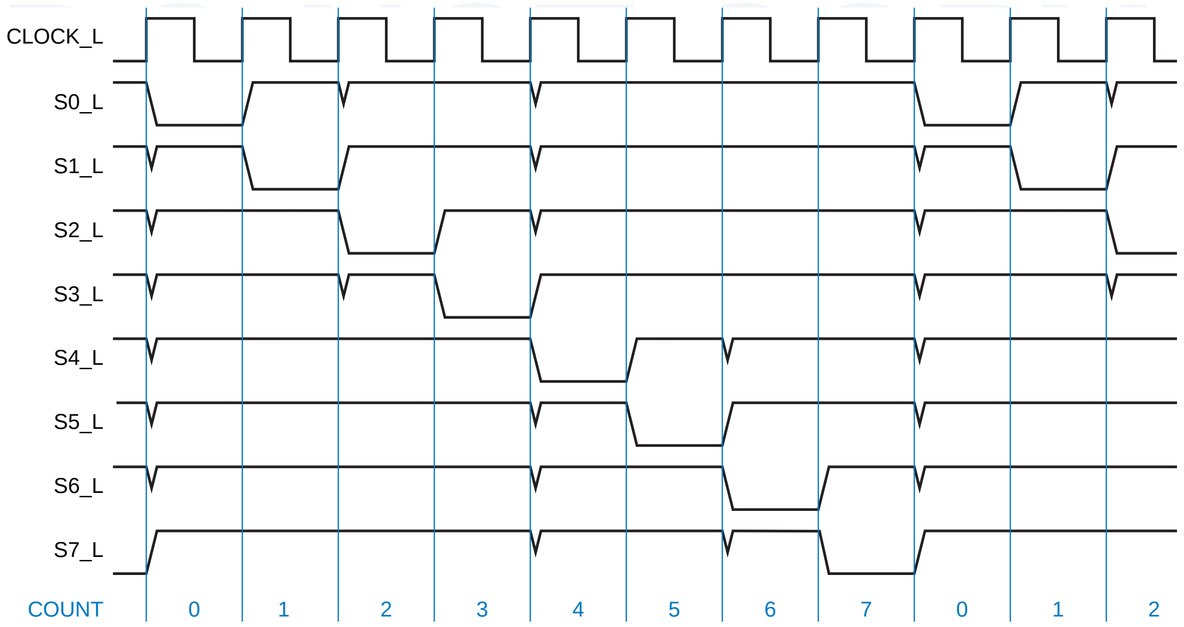


Figure 8-43 Timing diagram for a modulo-8 binary counter and decoder, showing decoding glitches.

multiple signal paths in a decoder like the ’138 have different delays; for example, the path from B to Y1_L is faster than the path from A to Y1_L. Thus, even if the input changes simultaneously from 011 to 100, the decoder may behave as if the input were temporarily 001, and the Y1_L output may have a glitch. In the present example, it can be shown that the glitches can occur in *any* realization of the binary decoder function; this problem is an example of a *function hazard*.

function hazard

In most applications, the decoder output signals portrayed in Figure 8-43 would be used as control inputs to registers, counters, and other edge-triggered devices (e.g., EN_L in a 74x377, LD_L in a 74x163, or ENP_L in a 74x169). In such a case, the decoding glitches in the figure are not a problem, since they occur *after* the clock tick. They are long gone before the next tick comes along, when the decoder outputs are sampled by other edge-triggered devices. However, the glitches *would* be a problem if they were applied to something like the S_L or R_L inputs of an $\overline{S}$-$\overline{R}$ latch. Likewise, using such potentially glitchy signals as clocks for edge-triggered devices is a definite no-no.

If necessary, one way to “clean up” the glitches in Figure 8-43 is to connect the ’138 outputs to another register that samples the stable decoded outputs on the next clock tick, as shown in Figure 8-44. Notice that the decoded outputs have been renamed to account for the 1-tick delay through the register. However, once you decide to pay for an 8-bit register, a less costly solution is to use an 8-bit “ring counter,” which provides glitch-free decoded outputs directly, as we’ll show in Section 8.5.6.

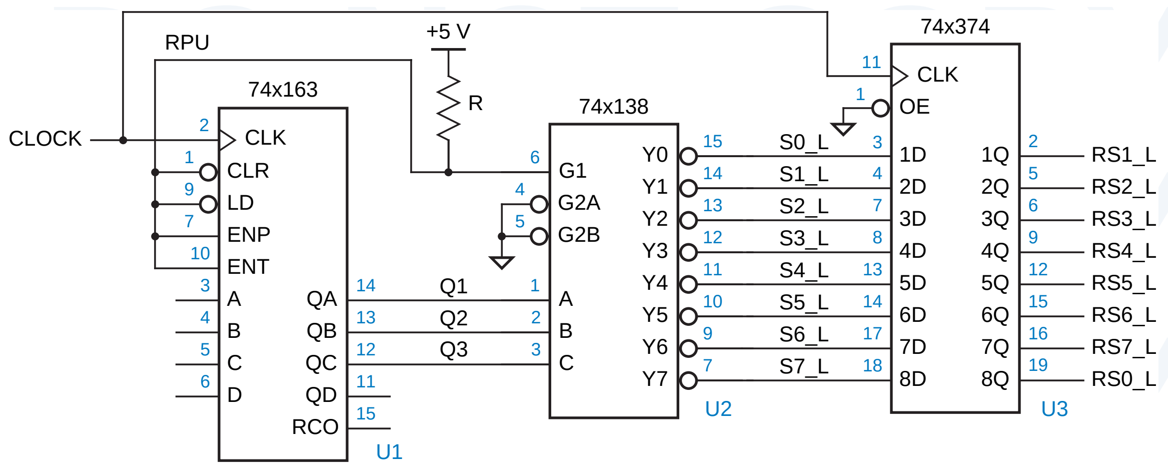


Figure 8-44 A modulo-8 binary counter and decoder with glitch-free outputs.

8.4.5 Counters in ABEL and PLDs

Binary counters are good candidates for ABEL- and PLD-based design, for several reasons:

- A large state machine can often be decomposed into two or more smaller state machines where one of the smaller machines is a binary counter that keeps track of how long the other machine should stay in a particular state. This may simplify both the conceptual design and the circuit design of the machine.
- Many applications require almost-binary-modulus counters with special requirements for initialization, state detection, or state skipping. For example, a counter in an elevator controller may skip state 13. Instead of using an off-the-shelf binary counter and extra logic for the special requirements, a designer can specify exactly the required functions in an ABEL program.
- Most standard MSI counters have only 4 bits, while a single 24-pin PLD can be used to create a binary counter with up to 10 bits.

The most popular MSI counter is the 74x163 4-bit binary counter, shown in Figure 8-31 on page 599. A glance at this figure shows that the excitation logic for this counter isn't exactly simple, especially considering its use of XNOR gates. Nevertheless, ABEL provides a very simple way of defining counter behavior, which we describe next.

Recall that ABEL uses the "+" symbol to specify integer addition. When two sets are "added" with this operator, each is interpreted as a binary number; the rightmost set element corresponds to the least significant bit of the number. Thus, the function of a 74x163 can be specified by the ABEL program in Table 8-12. When the counter is enabled, 1 is added to the current state.

Table 8-12 ABEL program for a 74x163-like 4-bit binary counter.

```
module Z74X163
title '4-bit Binary Counter'

" Input pins
CLK, !LD, !CLR, ENP, ENT        pin;
A, B, C, D                      pin;

" Output pins
QA, QB, QC, QD                  pin istype 'reg';
RCO                             pin istype 'com';

" Set definitions
INPUT = [D, C, B, A];
COUNT = [QD, QC, QB, QA];

equations

COUNT.CLK = CLK;

COUNT := !CLR & ( LD & INPUT
                # !LD & (ENT & ENP) & (COUNT + 1)
                # !LD & !(ENT & ENP) & COUNT);

RCO = (COUNT == [1,1,1,1]) & ENT;

end Z74X163
```

Table 8-13 MInimized equations for the 4-bit binary counter in Table 8-12.

```
QA := (CLR & LD & ENT & ENP & !QA           QD := (CLR & LD & ENT & ENP & !QD & QC & QB & QA
     # CLR & LD & !ENP & QA                      # CLR & !LD & D
     # CLR & LD & !ENT & QA                      # CLR & LD & QD & !QB
     # CLR & !LD & A);                           # CLR & LD & QD & !QC
                                                 # CLR & LD & !ENP & QD
QB := (CLR & LD & ENT & ENP & !QB & QA           # CLR & LD & !ENT & QD
     # CLR & LD & QB & !QA                       # CLR & LD & QD & !QA);
     # CLR & LD & !ENP & QB
     # CLR & LD & !ENT & QB                 RCO = (ENT & QD & QC & QB & QA);
     # CLR & !LD & B);

QC := (CLR & LD & ENT & ENP & !QC & QB & QA
     # CLR & LD & QC & !QA
     # CLR & LD & QC & !QB
     # CLR & LD & !ENP & QC
     # CLR & LD & !ENT & QC
     # CLR & !LD & C);
```

Table 8-13 shows the minimized logic equations that ABEL generates for the 4-bit counter. Notice that each more significant output bit requires one more product term. As a result, the size of counters that can be realized in a 16V8 or even a 20V8 is generally limited to five or six bits. Other devices, including the X-series PLDs and some CPLDs, contain an XOR structure that can realize larger counters without increasing product-term requirements.

Designing a specialized counting sequence in ABEL is much simpler than adapting a standard binary counter. For example, the ABEL program in Table 8-12 can be adapted to count in excess-3 sequence (Figure 8-38 on page 603) by changing the equations as follows:

```
COUNT := !CLR & ( LD & INPUT
       # !LD & (ENT & ENP) &
           ((COUNT==12) & 3) # ((COUNT!=12) & (COUNT + 1))
       # !LD & !(ENT & ENP) & COUNT);

RCO = (COUNT == 12) & ENT;
```

PLDs can be cascaded to obtain wider counters, by providing each counter stage with a carry output that indicates when it is about to roll over. There are two basic approaches to generating the carry output:

combinational carry output

- *Combinational*. The carry equation indicates that the counter is enabled and is currently in its last state before rollover. For a 5-bit binary up counter, we have

  ```
  COUT = CNTEN & Q4 & Q3 & Q2 & Q1 & Q0;
  ```

  Since CNTEN is included, this approach allows carries to be rippled through cascaded counters by connecting each COUT to the next CNTEN.

registered carry output

- *Registered*. The carry equation indicates that the counter is about to enter its last state before rollover. Thus, at the next clock tick, the counter enters this last state and the carry output is asserted. For a 5-bit binary up counter with load and clear inputs, we have

  ```
  COUT := !CLR & !LD & CNTEN
            & Q4 & Q3 & Q2 & Q1 & !Q0
        # !CLR * !LD * !CNTEN
            & Q4 & Q3 & Q2 & Q1 & Q0
        # !CLR & LD
            & D4 & D3 & D2 & D1 & D0;
  ```

The second approach has the advantage of producing COUT with less delay than the combinational approach. However, external gates are now required between stages, since the CNTEN signal for each stage should be the logical AND of the master count-enable signal and the COUT outputs of all lower-order counters. These external gates can be avoided if the higher-order counters have multiple enable inputs.

8.4.6 Counters in VHDL

Like ABEL, VHDL allows counters to be specified fairly easily. The biggest challenge in VHDL, with its strong type checking, is to get all of the signal types defined correctly and consistently.

Table 8-14 is a VHDL program for a 74x163-like binary counter. Notice that the program uses the `IEEE.std_logic_arith.all` library, which includes the `UNSIGNED` type, as we described in Section 5.9.6 on page 389. This library includes definitions of "+" and "-" operators that perform unsigned addition and subtraction on `UNSIGNED` operands. The counter program declares the counter input and output as `UNSIGNED` vectors and uses "+" to increment the counter value as required.

In the program, we defined an internal signal `IQ` to hold the counter value. We could have used `Q` directly, but then we'd have to declare its port type as `buffer` rather than `out`. Also, we could have defined the type of ports `D` and `Q` to be `STD_LOGIC_VECTOR`, but then we would have to perform type conversions inside the body of the process (see Exercise 8.33).

Table 8-14 VHDL program for a 74x163-like 4-bit binary counter.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity V74x163 is
    port ( CLK, CLR_L, LD_L, ENP, ENT: in STD_LOGIC;
            D: in UNSIGNED (3 downto 0);
            Q: out UNSIGNED (3 downto 0);
            RCO: out STD_LOGIC );
end V74x163;

architecture V74x163_arch of V74x163 is
signal IQ: UNSIGNED (3 downto 0);
begin
process (CLK, ENT, IQ)
  begin
    if (CLK'event and CLK='1') then
      if CLR_L='0' then IQ <= (others => '0');
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=15) and (ENT='1') then RCO <= '1';
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74x163_arch;
```

Table 8-15 VHDL architecture for counting in excess-3 order.

```
architecture V74xs3_arch of V74x163 is
signal IQ: UNSIGNED (3 downto 0);
begin
process (CLK, ENT, IQ)
  begin
    if CLK'event and CLK='1' then
      if CLR_L='0' then IQ <= (others => '0');
      elsif LD_L='0' then IQ <= D;
      elsif (ENT and ENP)='1' and (IQ=12) then IQ <= ('0','0','1','1');
      elsif (ENT and ENP)='1' then IQ <= IQ + 1;
      end if;
    end if;
    if (IQ=12) and (ENT='1') then RCO <= '1';
    else RCO <= '0';
    end if;
    Q <= IQ;
  end process;
end V74xs3_arch;
```

As in ABEL, specialized counting sequences can be specified very easily using behavioral VHDL code. For example, Table 8-15 modifies the 74x163-like counter to count in excess-3 sequence (3, … , 12, 3, …).

Unfortunately, some VHDL synthesis engines do not synthesize counters particularly well. In particular, they tend to synthesize the counting step using a binary adder with the counter value and a constant 1 as operands. This approach requires much more combinational logic than what we've shown for discrete counters, and is particularly wasteful in CPLDs and FPGAs containing T flip-flops, XOR gates, or other structures optimized for counters. In this case, a useful alternative is to write structural VHDL that is targeted to the cells available in a particular CPLD, FPGA, or ASIC technology.

For example, we can construct one bit-cell for a 74x163-like counter using the circuit in Figure 8-45. This circuit is designed to use serial propagation for the carry bits, so the same circuit can be used at any stage of an arbitrarily large counter, subject to fanout constraints on the common signals that drive all of the stages. The signals in the bit-cell have the following definitions:

CLK	(common) The clock input for all stages.
LDNOCLR	(common) Asserted if the counter's LD input is asserted and CLR is negated.
NOCLRORLD	(common) Asserted if the counter's CLR and LD inputs are both negated.
CNTENP	(common) Asserted if the counter's ENP input is asserted.
Di	(per cell) Load data input for cell *i*.

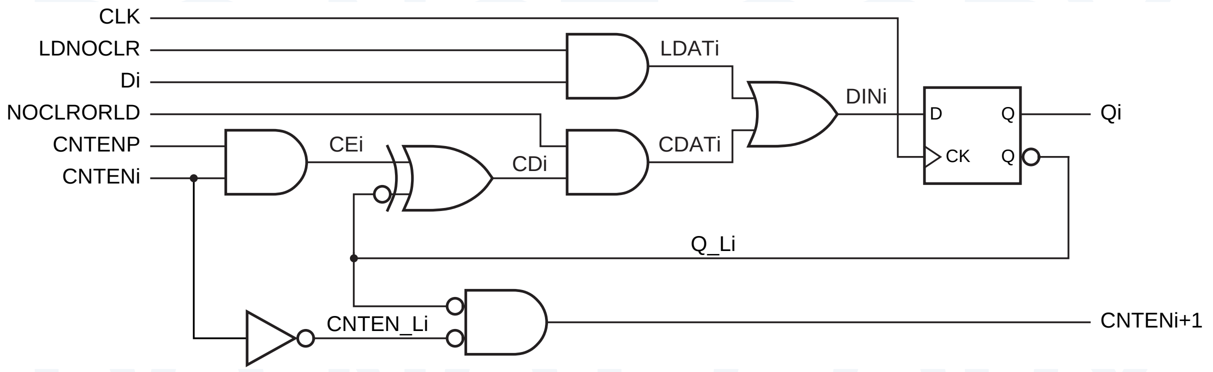


Figure 8-45 One bit-cell of a synchronous serial, 74x163-like counter.

CNTENi	(per cell) Serial count enable input for cell *i*.
CNTENi+1	(per cell) Serial count enable output for cell *i*.
Qi	(per cell) Counter output for cell *i*.

Table 8-16 is a VHDL program corresponding to the bit-cell in the figure. In the program, the D flip-flop component `Vdffqqn` is assumed to be already defined; it is similar to the D flip-flop in Table 8-6 with the addition of a QN (complemented) output. In an FPGA or ASIC design, a flip-flop component type would be chosen from the manufacturer's standard cell library.

Table 8-16 VHDL program for counter cell of Figure 8-45.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity syncsercell is
  port( CLK, LDNOCLR, NOCLRORLD, CNTENP, D, CNTEN: in STD_LOGIC;
        CNTENO, Q: out STD_LOGIC );
end syncsercell;

architecture syncsercell_arch of syncsercell is
component Vdffqqn
  port( CLK, D: in STD_LOGIC;
        Q, QN: out STD_LOGIC );
end component;
signal LDAT, CDAT, DIN, Q_L: STD_LOGIC;
begin
  LDAT <= LDNOCLR and D;
  CDAT <= NOCLRORLD and ((CNTENP and CNTEN) xor not Q_L);
  DIN  <= LDAT or CDAT;
  CNTENO <= (not Q_L) and CNTEN;
  U1: Vdffqqn port map (CLK, DIN, Q, Q_L);
end syncsercell_arch;
```

A MATTER OF STYLE

Note that Table 8-16 uses a combination of dataflow and structural VHDL styles. It could have been written completely structurally, for example using an ASIC manufacturer's gate component definitions, to guarantee that the synthesized circuit conforms exactly to Figure 8-45. However, most synthesis engines can do a good job of picking the best gate realization for the simple signal assignments used here.

Table 8-17 shows how to create an 8-bit synchronous serial counter using the cell defined previously. The first two assignments in the architecture body synthesize the common LDNOCLR and NOCLRORLD signals. The next two statements handle boundary condition for the serial count-enable chain. Finally, the `generate` statement (introduced on page 415) instantiates eight 1-bit counter cells and hooks up the count-enable chain as required.

It should be clear that a larger or smaller counter can be created simply by changing a few definitions in the program. You can put VHDL's `generic` statement to good use here to allow you to change the counter's size with a one-line change (see Exercise 8.35).

Table 8-17 VHDL program for an 8-bit 74x163-like synchronous serial counter.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity V74x163s is
  port( CLK, CLR_L, LD_L, ENP, ENT: in STD_LOGIC;
        D: in STD_LOGIC_VECTOR (7 downto 0);
        Q: out STD_LOGIC_VECTOR (7 downto 0);
        RCO: out STD_LOGIC );
end V74x163s;

architecture V74x163s_arch of V74x163s is
component syncsercell
  port( CLK, LDNOCLR, NOCLRORLD, CNTENP, D, CNTEN: in STD_LOGIC;
        CNTENO, Q: out STD_LOGIC );
end component;
signal LDNOCLR, NOCLRORLD: STD_LOGIC; -- common signals
signal SCNTEN: STD_LOGIC_VECTOR (8 downto 0); -- serial count-enable inputs
begin
  LDNOCLR <= (not LD_L) and CLR_L; -- create common load and clear controls
  NOCLRORLD <= LD_L and CLR_L;
  SCNTEN(0) <= ENT; -- serial count-enable into the first stage
  RCO <= SCNTEN(8);  -- RCO is equivalent to final count-enable output
  g1: for i in 0 to 7 generate    -- generate the eight syncsercell stages
        U1: syncsercell port map ( CLK, LDNOCLR, NOCLRORLD, ENP, D(i), SCNTEN(i),
                                   SCNTEN(i+1), Q(i));
  end generate;
end V74x163s_arch;
```

8.5 Shift Registers

8.5.1 Shift-Register Structure

shift register

serial input

serial output

A *shift register* is an *n*-bit register with a provision for shifting its stored data by one bit position at each tick of the clock. Figure 8-46 shows the structure of a serial-in, serial-out shift register. The *serial input*, SERIN, specifies a new bit to be shifted into one end at each clock tick. This bit appears at the *serial output*, SEROUT, after *n* clock ticks, and is lost one tick later. Thus, an *n*-bit serial-in, serial-out shift register can be used to delay a signal by *n* clock ticks.

serial-in, parallel-out shift register

serial-to-parallel conversion

A *serial-in, parallel-out shift register,* shown in Figure 8-47, has outputs for all of its stored bits, making them available to other circuits. Such a shift register can be used to perform *serial-to-parallel conversion*, as explained later in this section.

parallel-in, serial-out shift register

Conversely, it is possible to build a *parallel-in, serial-out shift register.* Figure 8-48 shows the general structure of such a device. At each clock tick, the register either loads new data from inputs 1D–ND, or it shifts its current contents, depending on the value of the LOAD/SHIFT control input (which could be named LOAD or SHIFT_L). Internally, the device uses a 2-input multiplexer on each flip-flop's D input to select between the two cases. A parallel-in, serial-out shift register can be used to perform *parallel-to-serial conversion*, as explained later in this section.

parallel-to-serial conversion

parallel-in, parallel-out shift register

By providing outputs for all of the stored bits in a parallel-in shift register, we obtain the *parallel-in, parallel-out shift register* shown in Figure 8-49. Such a device is general enough to be used in any of the applications of the previous shift registers.

Figure 8-46 Structure of a serial-in, serial-out shift register.

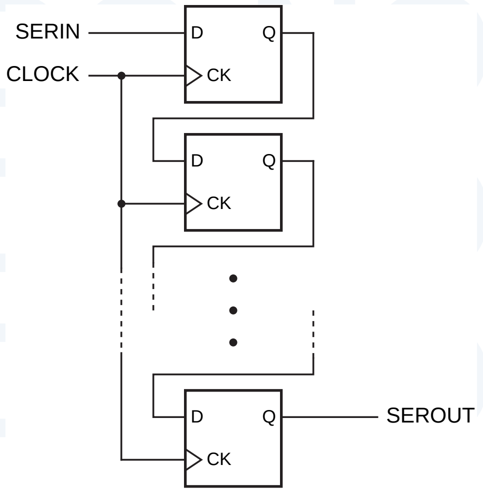
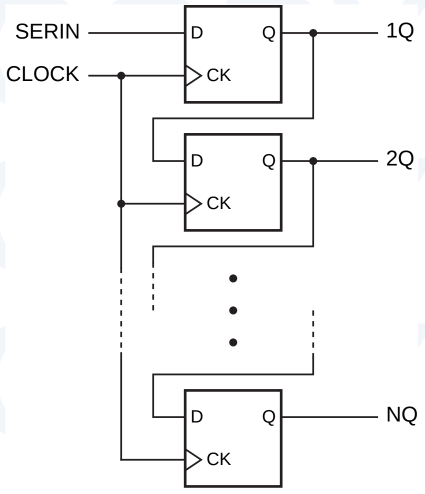


Figure 8-47 Structure of a serial-in, parallel-out shift register.



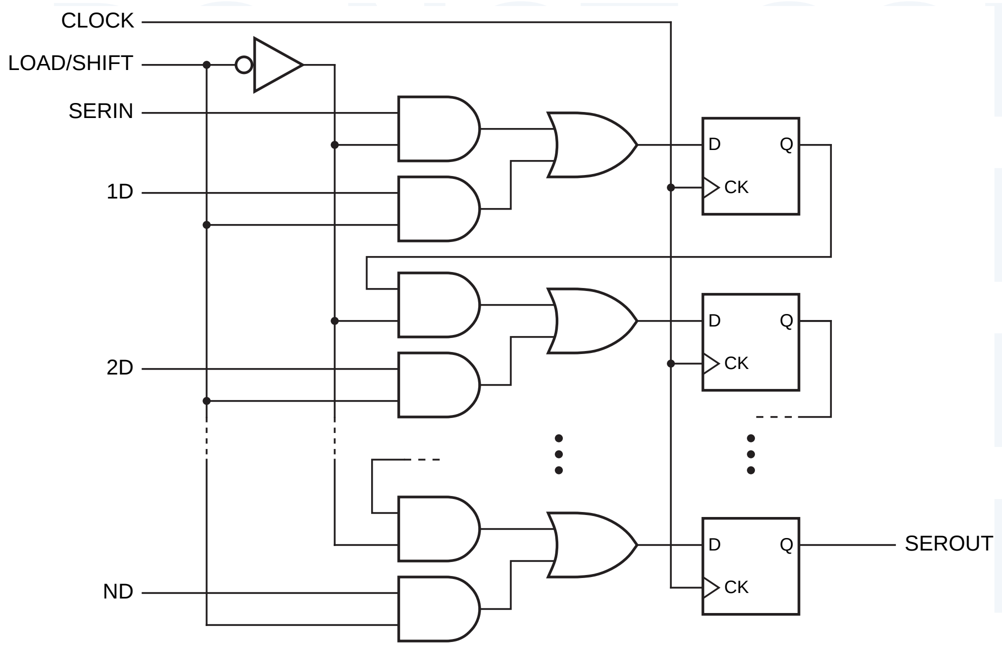
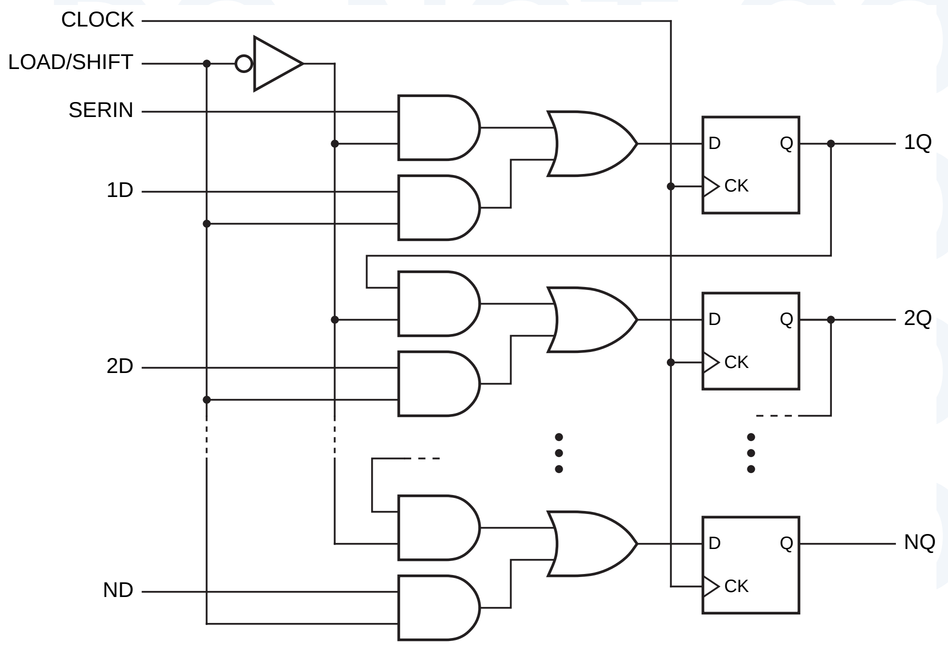


Figure 8-48 Structure of a parallel-in, serial-out shift register.

Figure 8-49 Structure of a parallel-in, parallel-out shift register.



8.5.2 MSI Shift Registers

74x164

Figure 8-50 shows logic symbols for three popular MSI 8-bit shift registers. The *74x164* is a serial-in, parallel-out device with an asynchronous clear input (CLR_L). It has two serial inputs that are ANDed internally. That is, both SERA and SERB must be 1 for a 1 to be shifted into the first bit of the register.

74x166

The *74x166* is a parallel-in, serial-out shift register, also with an asynchronous clear input. This device shifts when SH/LD is 1, and loads new data otherwise. The '166 has an unusual clocking arrangement called a "gated clock" (see also Section 8.8.2); it has two clock inputs that are connected to the internal flip-flops as shown in Figure 8-50(c). The designers of the '166 intended for CLK to be connected to a free-running system clock, and for CLKINH to be asserted to inhibit CLK, so that neither shifting nor loading occurs on the next clock tick, and the current register contents are held. However, for this to work, CLKINH must be changed only when CLK is 1; otherwise, undesired clock edges occur on the internal flip-flops. A much safer way of obtaining a "hold" function is employed in the next devices that we discuss.

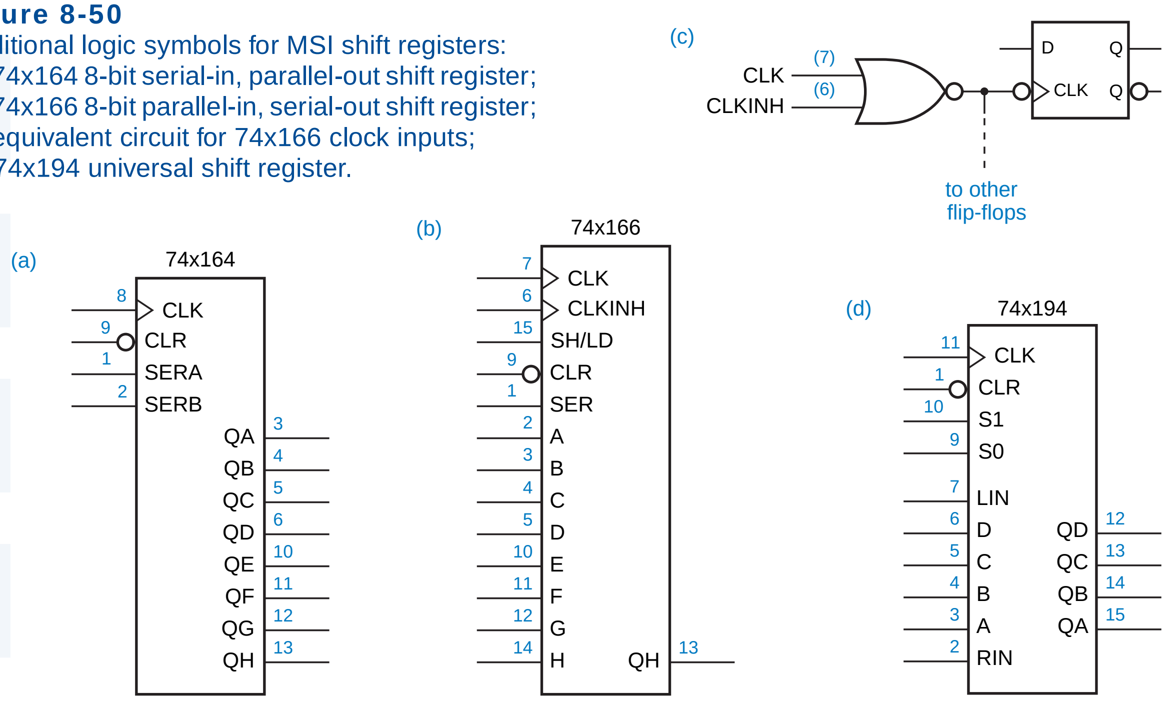
74x194

unidirectional shift register

bidirectional shift register

The *74x194* is an MSI 4-bit bidirectional, parallel-in, parallel-out shift register. Its logic diagram is shown in Figure 8-51. The shift registers that we've studied previously are called *unidirectional shift registers* because they shift in only one direction. The '194 is a *bidirectional shift register* because its contents may be shifted in either of two directions, depending on a control input. The two

Figure 8-50
Traditional logic symbols for MSI shift registers:
(a) 74x164 8-bit serial-in, parallel-out shift register;
(b) 74x166 8-bit parallel-in, serial-out shift register;
(c) equivalent circuit for 74x166 clock inputs;
(d) 74x194 universal shift register.



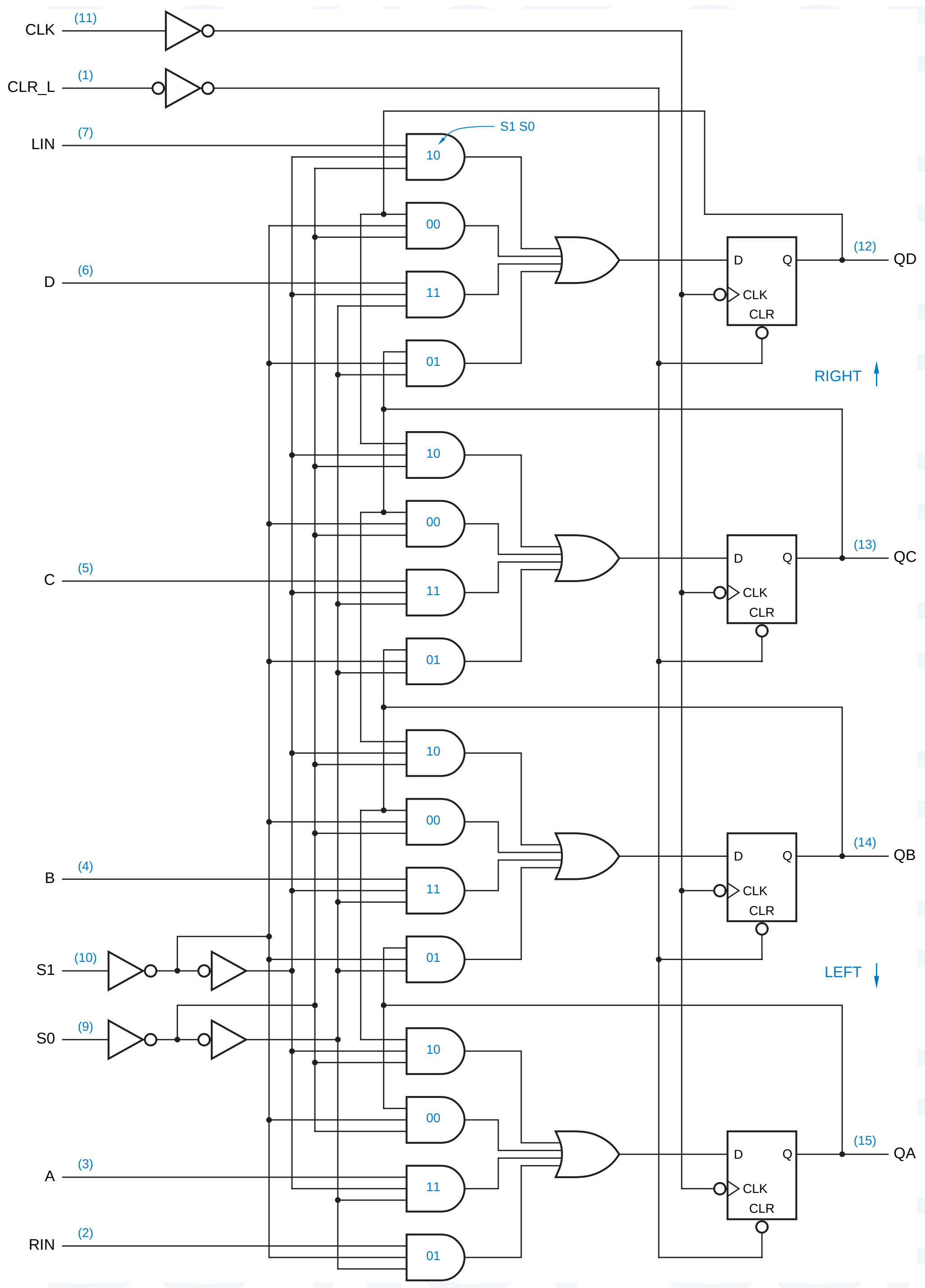


Figure 8-51 Logic diagram for the 74x194 4-bit universal shift register, including pin numbers for a standard 16-pin dual in-line package.

Table 8-18 Function table for the 74x194 4-bit universal shift register.

	Inputs		Next state			
Function	S1	S0	***QA****	***QB****	***QC****	***QD****
Hold	0	0	QA	QB	QC	QD
Shift right	0	1	RIN	QA	QB	QC
Shift left	1	0	QB	QC	QD	LIN
Load	1	1	A	B	C	D

left
right

directions are called “left” and “right,” even though the logic diagram and the logic symbol aren’t necessarily drawn that way. In the ’194, *left* means “in the direction from QD to QA,” and *right* means “in the direction from QA to QD.” Our logic diagram and symbol for the ’194 are consistent with these names if you rotate them 90° clockwise.

Table 8-18 is a function table for the 74x194. This function table is highly compressed, since it does not contain columns for most of the inputs (A–D, RIN, LIN) or the current state QA–QD. Still, by expressing each next-state value as a function of these implicit variables, it completely defines the operation of the ’194 for all 2^{12} possible combinations of current state and input, and it sure beats a 4,096-row table!

Note that the ’194’s LIN (left-in) input is conceptually located on the “right-hand” side of the chip, but it is the serial input for *left* shifts. Likewise, RIN is the serial input for right shifts.

The ’194 is sometimes called a *universal* shift register because it can be made to function like any of the less general shift register types that we’ve discussed (e.g., unidirectional; serial-in, parallel-out; parallel-in, serial-out). In fact, many of our design examples in the next few subsections contain ’194s configured to use just a subset of their available functions.

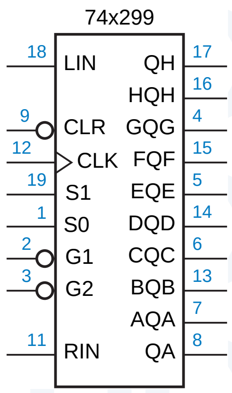


Figure 8-52 Traditional logic symbol for the 74x299.

The *74x299* is an 8-bit universal shift register in a 20-pin package; its symbol and logic diagram are given in Figures 8-52 and 8-53. The ’299’s functions and function table are similar to the ’194’s, as shown in Table 8-19. To save pins, the ’299 uses bidirectional three-state lines for input and output, as shown in the logic diagram. During load operations (S1 S0 = 11), the three-state

Table 8-19 Function table for a 74x299 8-bit universal shift register.

	Inputs		Next state							
Function	S1	S0	***QA****	***QB****	***QC****	***QD****	***QE****	***QF****	***QG****	***QH****
Hold	0	0	QA	QB	QC	QD	QE	QF	QG	QH
Shift right	0	1	RIN	QA	QB	QC	QD	QE	QF	QG
Shift left	1	0	QB	QC	QD	QE	QF	QG	QH	LIN
Load	1	1	AQA	BQB	CQC	DQD	EQE	FQF	GQG	HQH

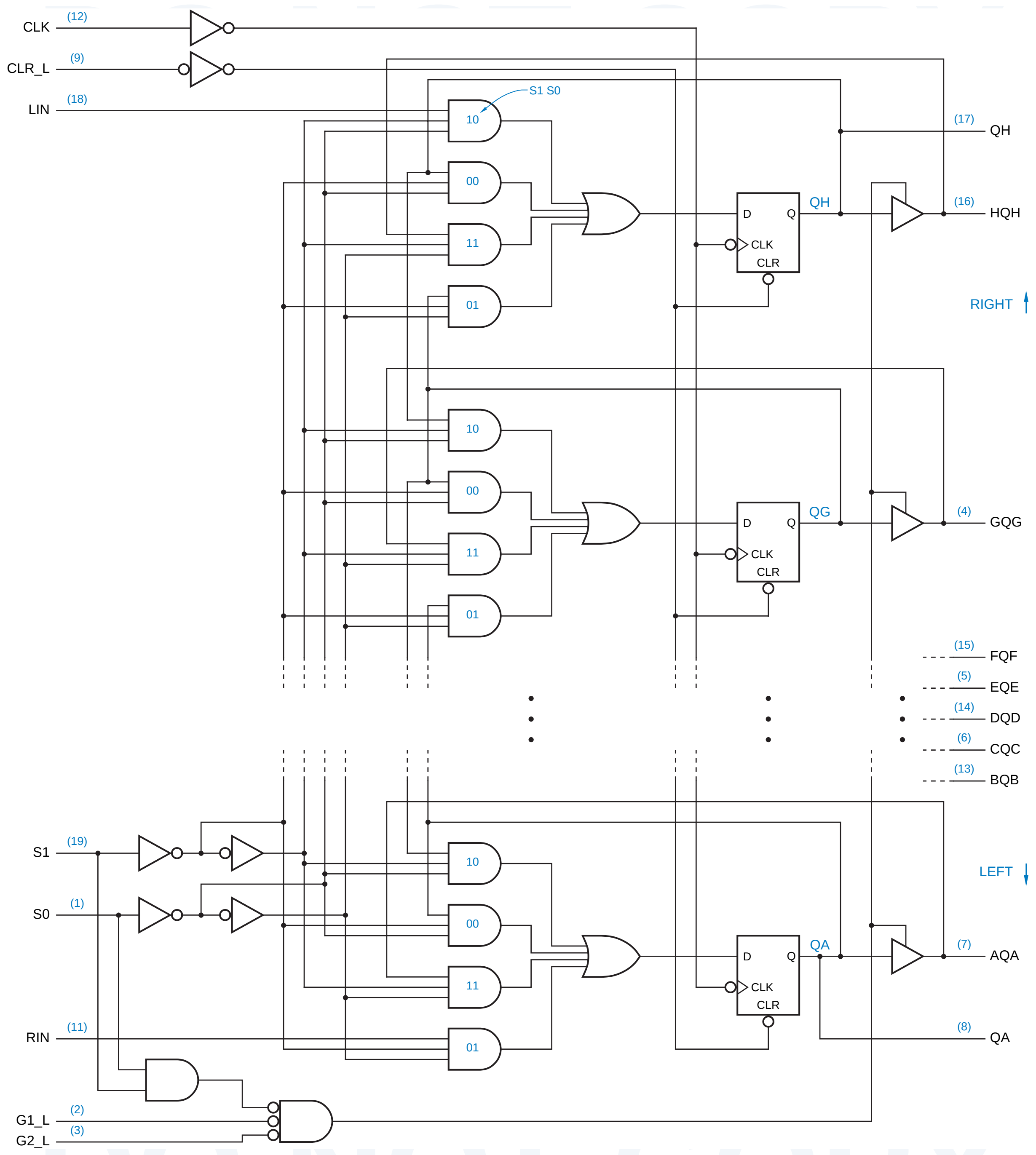


Figure 8-53 Logic diagram for the 74x299 8-bit universal shift register, including pin numbers for a standard 20-pin dual in-line package.

drivers are disabled and data is loaded through the AQA–HQH pins. At other times, the stored bits are driven onto these same pins if G1_L and G2_L are asserted. The leftmost and rightmost stored bits are available at all times on separate output-only pins, QA and QH.

8.5.3 The World's Biggest Shift-Register Application

The most common application of shift registers is to convert parallel data into serial format for transmission or storage, and to convert serial data back to parallel format for processing or display (see Section 2.16.1). The most common example of serial data transmission, one that you almost certainly take part in every day, is in *digital telephony.*

digital telephony

For years, TPCs (The Phone Companies) have been installing digital switching equipment in their central offices (*COs*). Most home phones have a two-wire analog connection to the central office. However, an analog-to-digital converter samples the analog voice signal 8,000 times per second (once every 125 μs) when it enters the CO, and produces a corresponding sequence of 8,000 8-bit bytes representing the sign and amplitude of the analog signal at each sampling point. Subsequently, your voice is transmitted digitally on 64-Kbps *serial channels* throughout the phone network, until it is converted back to an analog signal by a digital-to-analog converter at the far-end CO.

CO

serial channel

The 64 Kbps bandwidth required by a single digital voice signal is far *less* than can be obtained on a single digital signal line or switched by digital ICs. Therefore most digital telephone equipment *multiplexes* many 64-Kbps channels onto a single wire, saving both wires and digital ICs for switching. In the next subsection, we show how 32 channels can be processed by a handful of MSI chips; and these chips could be easily integrated into a single CPLD. This is a classic example of a *space/time trade-off* in digital design—by running the chips faster, you can accomplish a larger task with fewer chips. Indeed, this is the main reason that the telephone network has "gone digital."

multiplex

space/time trade-off

I STILL DON'T KNOW

ISDN (Integrated Services Digital Network) technology was developed in the late 1980s to extend full-duplex 144-kbps serial digital channels to home phones. The idea was to carry two 64-Kbps voice conversations plus a 16-Kbps control channel on a single pair of wires, thus increasing the capacity of installed wiring.

In the first edition of this book, we noted that delays in ISDN deployment had led some people in the industry to rename it "Imaginary Services Delivered Nowhere." In the mid-1990s, ISDN finally took off in the U.S., but it was deployed not so much to carry voice as to provide "high-speed" connections to the Internet.

Unfortunately for TPCs, the growth in ISDN was cut short first by the deployment of inexpensive 56-Kbps analog modems and later by the growing availability of very high-speed connections (160 Kbps to 2 Mbps or higher) using newer DSL (Digital Subscriber Line) and cable-modem technologies.

8.5.4 Serial/Parallel Conversion

A typical application of serial data transfer between two modules (possibly in a piece of CO switching equipment) is shown in Figure 8-54. Three signals are normally connected between a source module and a destination module to accomplish the transfer:

- *Clock.* The clock signal provides the timing reference for transfers, defining the time to transfer one bit. In systems with just two modules, the clock may be part of control circuits located on the source module as shown. In larger systems, the clock may be generated at a common point and distributed to all of the modules.
- *Serial data.* The data itself is transmitted on a single line.
- *Synchronization.* A *synchronization pulse* (or *sync pulse*) provides a reference point for defining the data format, such as the beginning of a byte or word in the serial data stream. Some systems omit this signal and instead use a unique pattern on the serial data line for synchronization.

synchronization pulse
sync pulse

The general timing characteristics of these signals in a typical digital telephony application are shown in Figure 8-55(a). The CLOCK signal has a frequency of 2.048 MHz to allow the transmission of $32 \times 8{,}000$ 8-bit bytes per

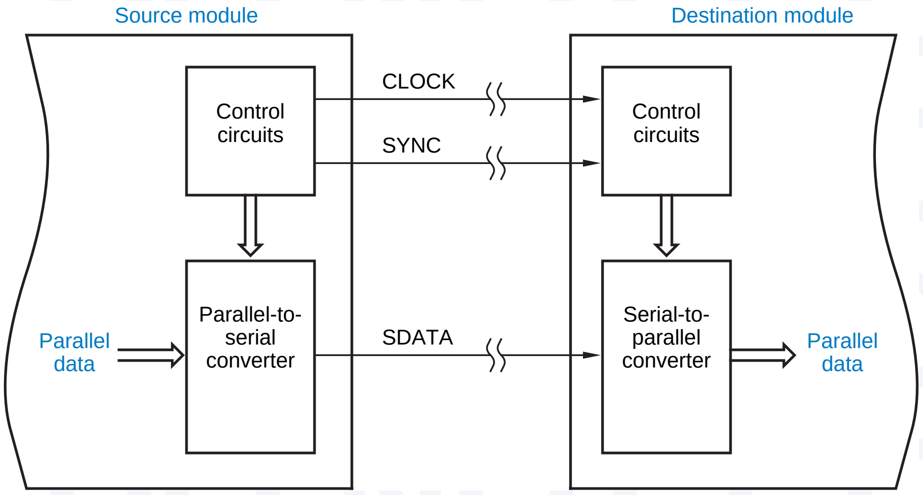


Figure 8-54 A system that transmits data serially between modules.

THE NATION'S CLOCK

Believe it or not, in the phone network, a very precise 8-KHz clock is generated in St. Louis and distributed throughout the U.S.! The clock signal that is distributed in a particular piece of local CO equipment is normally derived from the national clock. For example, the 2.048-MHz clock in this section's example could be derived by a phase-locked loop circuit that multiplies the national clock frequency by 256.

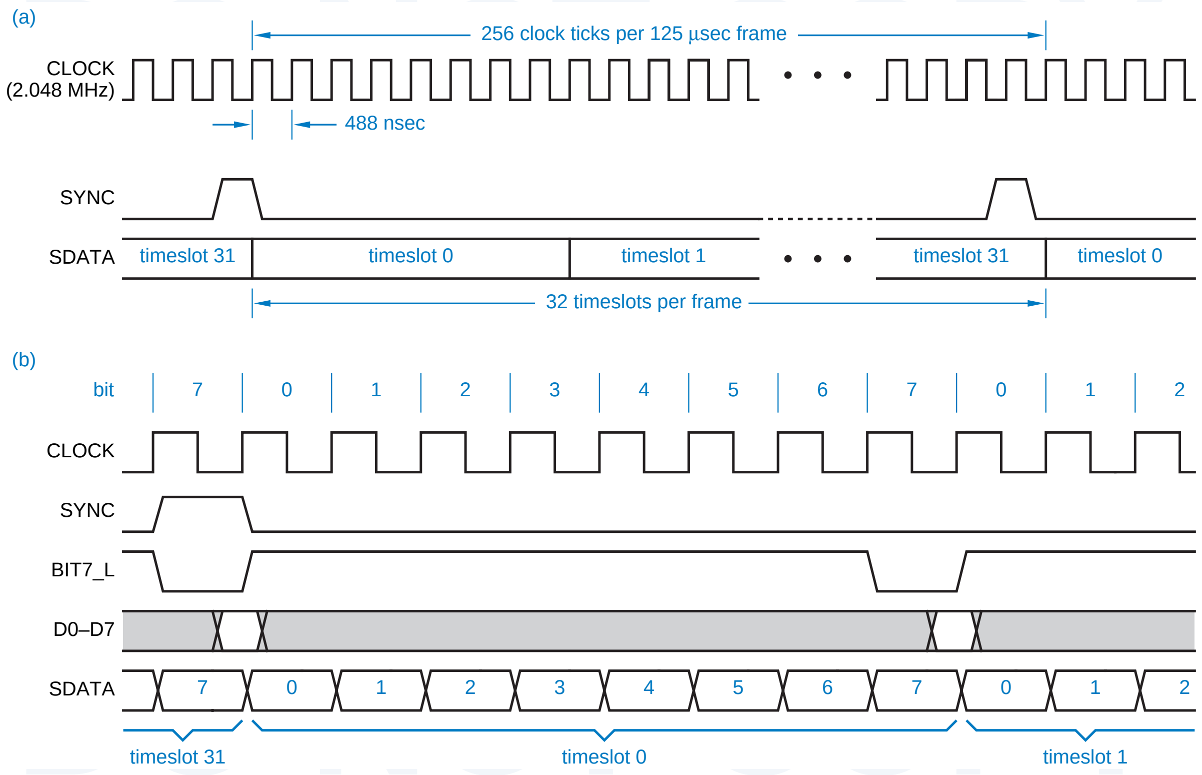


Figure 8-55 Timing diagram for parallel-to-serial conversion: (a) a complete frame; (b) one byte at the beginning of the frame.

second. The 1-bit-wide pulse on the SYNC signal identifies the beginning of a 125-μs interval called a *frame*. A total of 256 bits are transmitted on SDATA during this interval, which is divided into 32 *timeslots* containing eight bits each. Each timeslot carries one digitally encoded voice signal. Both timeslot numbers and bit positions within a timeslot are located relative to the SYNC pulse.

frame
timeslot

Figure 8-56 shows a circuit that converts parallel data to the serial format of Figure 8-55(a), with detailed timing shown in (b). Two 74x163 counters are wired as a free-running modulo-256 counter to define the frame. The five high-order and three low-order counter bits are the timeslot number and bit number, respectively.

WHICH BIT FIRST? Most real serial links for digitized voice actually transmit bit 7 first, because this is the first bit generated by the analog-to-digital converter that digitizes the voice signal. However, to simplify our examples, we transmit bit 0 first so that counter state equals bit number.

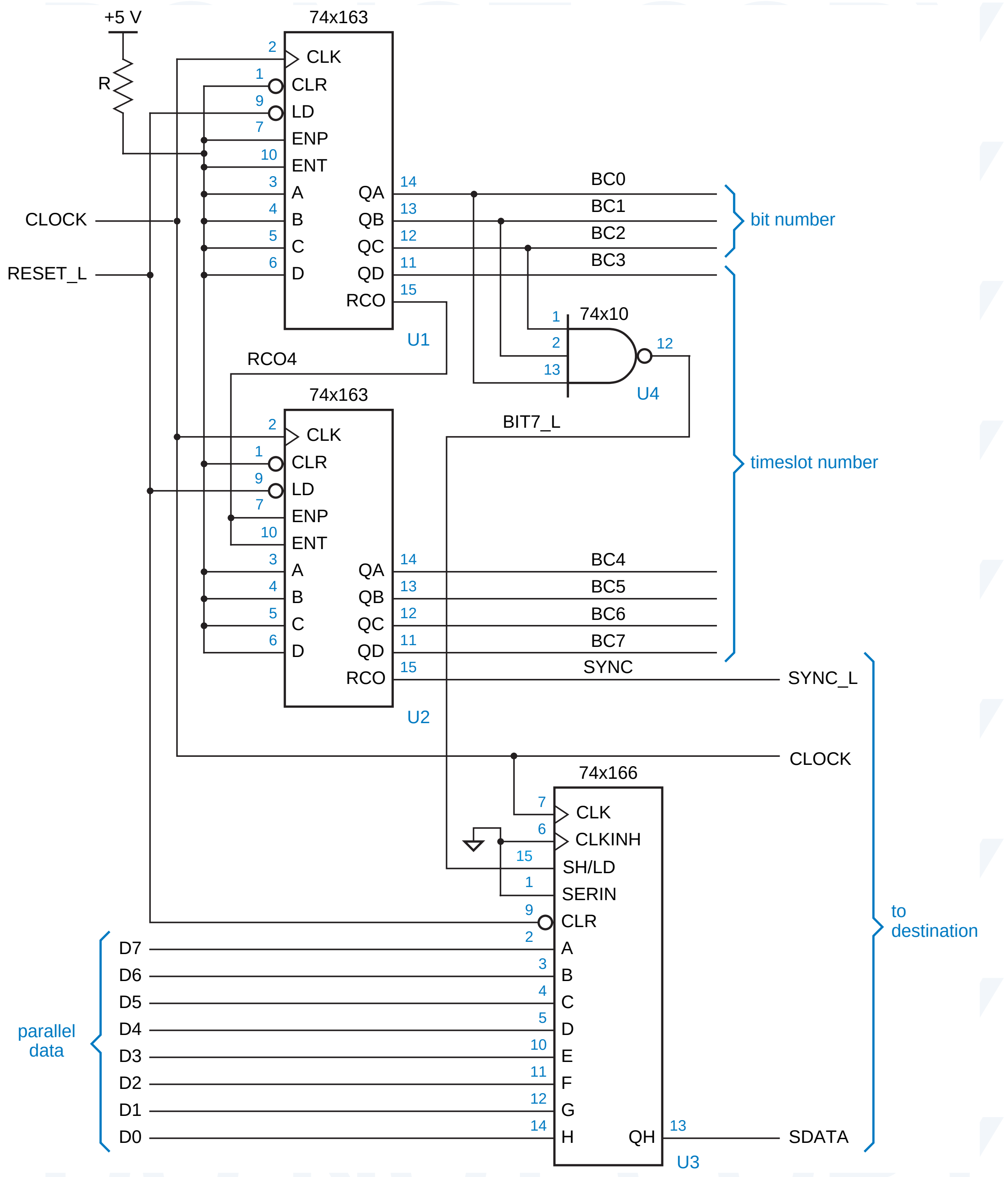


Figure 8-56 Parallel-to-serial conversion using a parallel-in shift register.

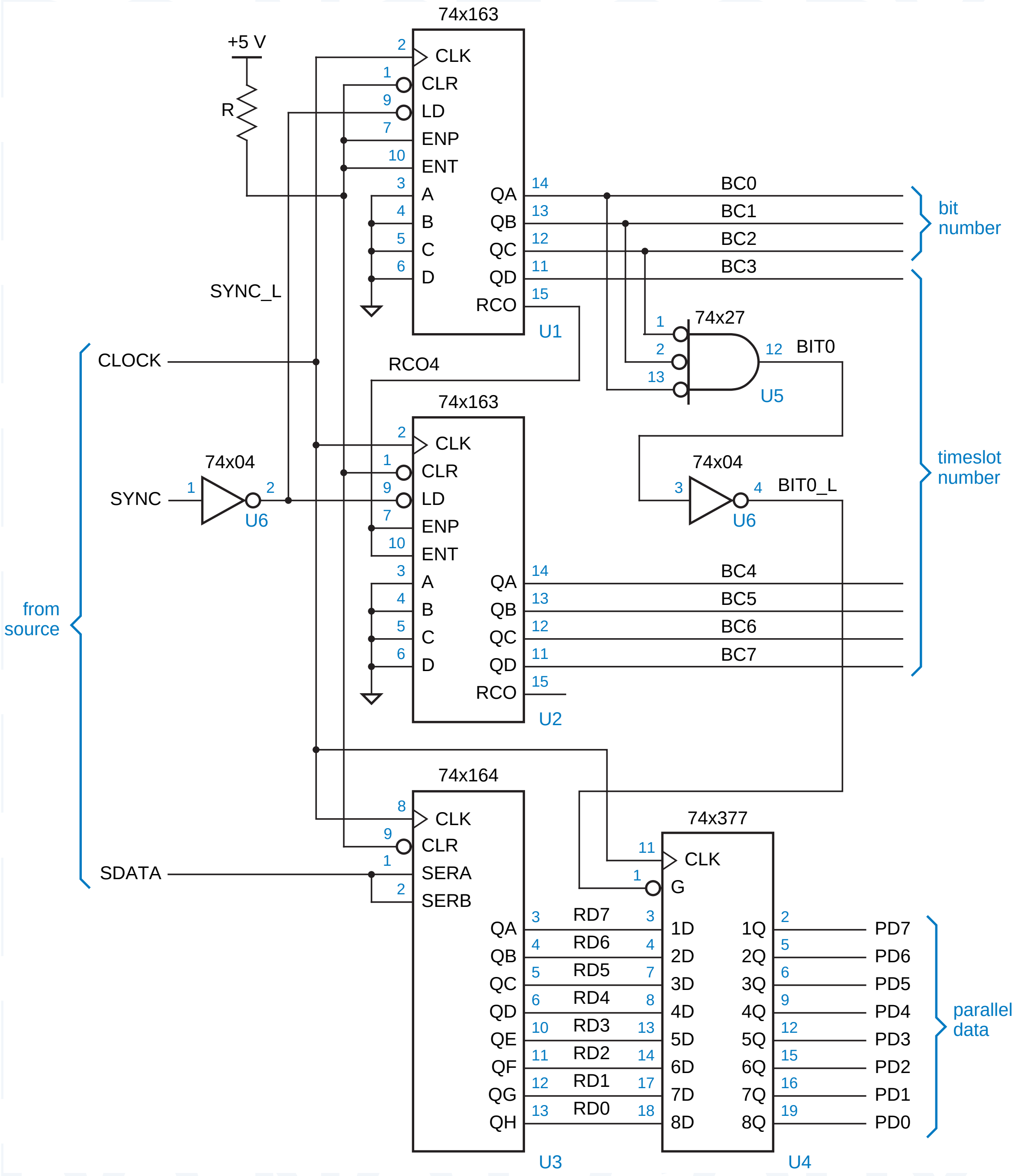


Figure 8-57 Serial-to-parallel conversion using a parallel-out shift register.

A 74x166 parallel-in shift register performs the parallel-to-serial conversion. Bit 0 of the parallel data (D0–D7) is connected to the ’166 input closest to the SDATA output, so bits are transmitted serially in the order 0 through 7.

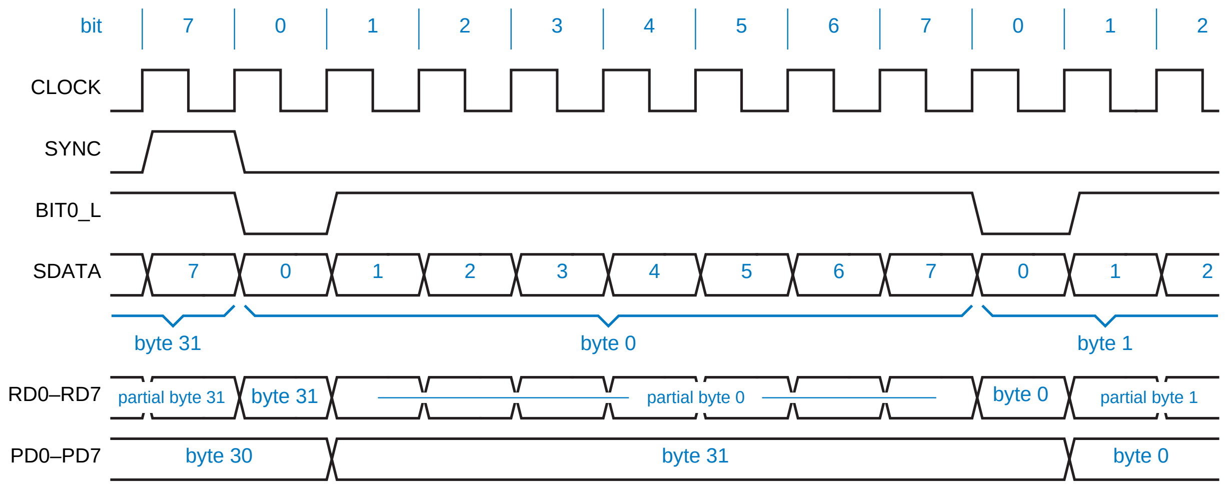
During bit 7 of each timeslot, the BIT7_L signal is asserted, which causes the ’166 to be loaded with D0–D7. The value of D0–D7 is irrelevant except during the setup- and hold-time window around the clock edge on which the ’166 is loaded, as shown by shading in the timing diagram. This leaves open the possibility that the parallel data bus could be used for other things at other times (see Exercise 8.36).

A destination module can convert the serial data back into parallel format using the circuit of Figure 8-57. A modulo-256 counter built from a pair of ’163s is used to reconstruct the timeslot and bit numbers. Although SYNC is asserted during state 255 of the counter on the source module, SYNC loads the destination module’s counter with 0 so that both counters go to 0 on the same clock edge. The counter’s high-order bits (timeslot number) are not used in the figure, but they may be used by other circuits in the destination module to identify the byte from a particular timeslot on the parallel data bus (PD0–PD7).

Figure 8-58 shows detailed timing for the serial-to-parallel conversion circuit. A complete received byte is available at parallel output of the 74x164 shift register during the clock period following the reception of the last bit (7) of the byte. The parallel data in this example is *double-buffered* —once it is fully received, it is transferred into a 74x377 register, where it is available on PD0–PD7 for eight full clock periods until the next byte is fully received. The BIT0_L signal enables the ’377 to be loaded at the proper time. Additional registers and decoding could be provided to load the byte from each timeslot into a different register, making each byte available for 125 μs (see Section 8.38).

double-buffered data

Figure 8-58 Timing diagram for serial-to-parallel conversion.



LITTLE ENDIANS AND BIG ENDIANS

At one point in the evolution of digital systems, the choice of which bit or byte to transmit first was a religious issue. In a famous article on the subject, “On Holy Wars and a Plea for Peace” (*Computer*, October 1981, pp. 48–54), Danny Cohen described the differences in bit- and byte-ordering conventions and the havoc that could be (and now has been) wrought as a result.

A firm standard was never established, so that today there are some popular computer systems (such as IBM-compatible PCs) that transmit or number the low-order byte of a 32-bit word first, and others (such as Apple Macintosh computers) that transmit or number the high-order byte first. Following Cohen’s nomenclature, people refer to these conventions as “Little Endian” and “Big Endian,” respectively, and talk about “endianness” as if it were actually a word.

Once the received data is in parallel format, it can easily be stored or modified by other digital circuits; we’ll give examples in Section 11.1.6. In digital telephony, the received parallel data is converted back into an analog voltage that is filtered and transmitted to an earpiece or speaker for 125 μs, until the next voice sample arrives.

8.5.5 Shift-Register Counters

shift-register counter

Serial/parallel conversion is a “data” application, but shift registers have “non-data” applications as well. A shift register can be combined with combinational logic to form a state machine whose state diagram is cyclic. Such a circuit is called a *shift-register counter.* Unlike a binary counter, a shift-register counter does not count in an ascending or descending binary sequence, but it is useful in many “control” applications nonetheless.

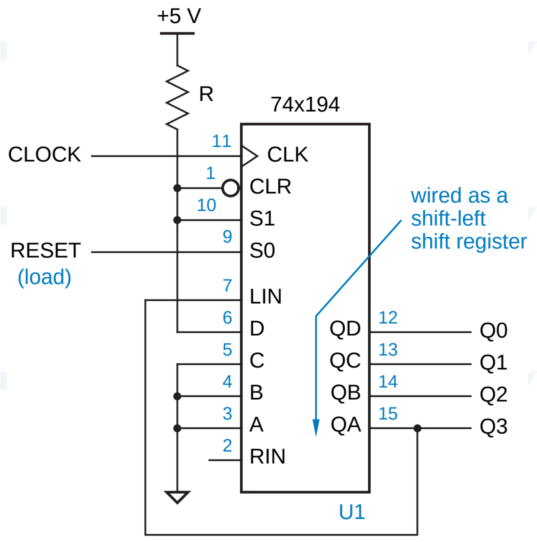


Figure 8-59 Simplest design for a four-bit, four-state ring counters with a single circulating 1.

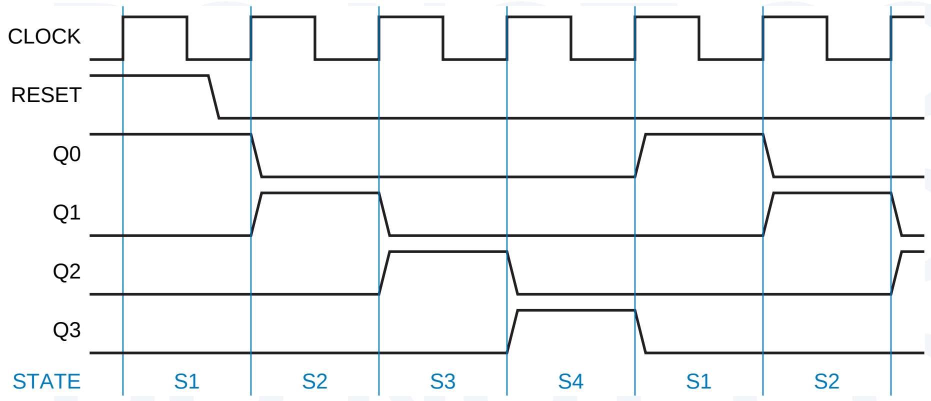


Figure 8-60
Timing diagram for a 4-bit ring counter.

8.5.6 Ring Counters

ring counter

The simplest shift-register counter uses an *n*-bit shift register to obtain a counter with *n* states, and is called a *ring counter*. Figure 8-59 is the logic diagram for a 4-bit ring counter. The 74x194 universal shift register is wired so that it normally performs a left shift. However, when RESET is asserted, it loads 0001 (refer to the '194's function table, Table 8-18 on page 618). Once RESET is negated, the '194 shifts left on each clock tick. The LIN serial input is connected to the "leftmost" output, so the next states are 0010, 0100, 1000, 0001, 0010, …. Thus, the counter visits four unique states before repeating. A timing diagram is shown in Figure 8-60. In general, an *n*-bit ring counter visits *n* states in a cycle.

The ring counter in Figure 8-59 has one major problem—it is not robust. If its single 1 output is lost due to a temporary hardware problem (e.g., noise), the counter goes to state 0000 and stays there forever. Likewise, if an extra 1 output is set (i.e., state 0101 is created), the counter will go through an incorrect cycle of states and stay in that cycle forever. These problems are quite evident if we draw the *complete* state diagram for the counter circuit, which has 16 states. As shown in Figure 8-61, there are 12 states that are not part of the normal counting cycle. If the counter somehow gets off the normal cycle, it stays off it.

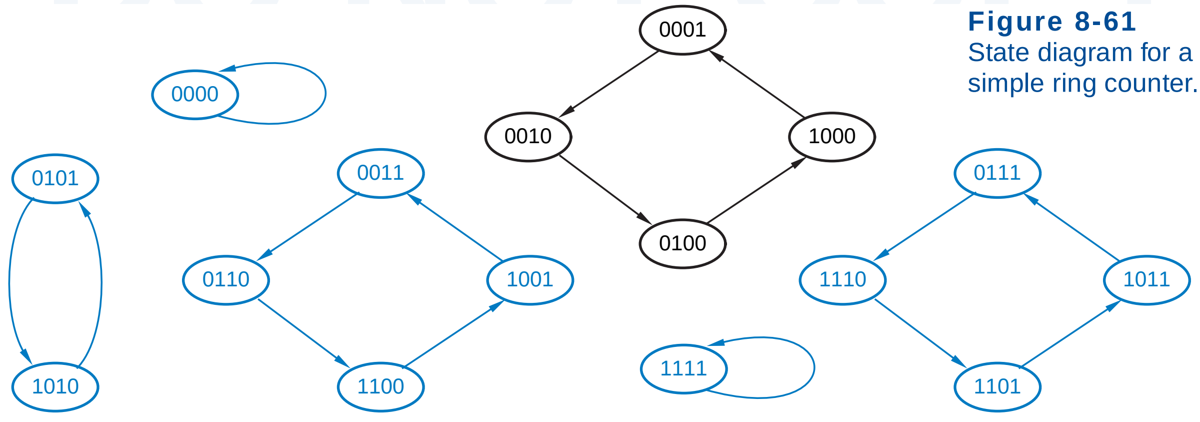


Figure 8-61
State diagram for a simple ring counter.

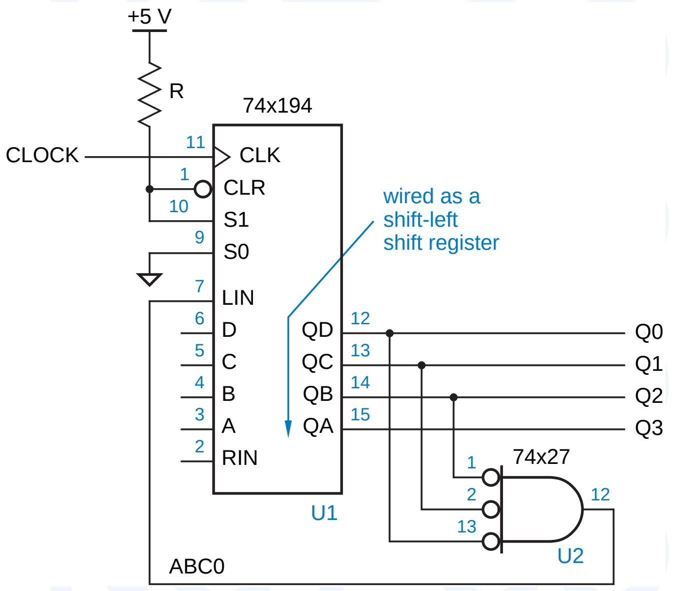


Figure 8-62 Self-correcting four-bit, four-state ring counter with a single circulating 1.

self-correcting counter

A *self-correcting counter* is designed so that all abnormal states have transitions leading to normal states. Self-correcting counters are desirable for the same reason that we use a minimal-risk approach to state assignment in Section 7.4.3: If something unexpected happens, a counter or state machine should go to a "safe" state.

self-correcting ring counter

A *self-correcting ring counter* circuit is shown in Figure 8-62. The circuit uses a NOR gate to shift a 1 into LIN only when the three least significant bits are 0. This results in the state diagram in Figure 8-63; all abnormal states lead back

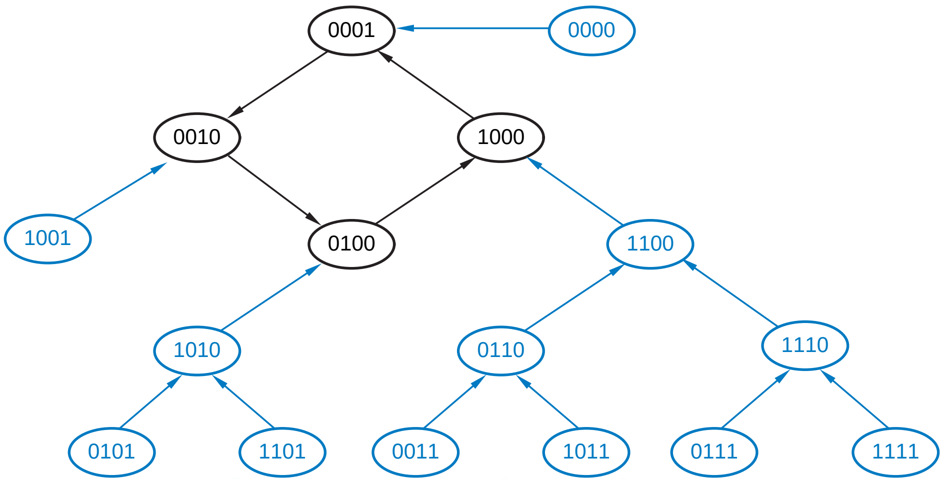


Figure 8-63 State diagram for a self-correcting ring counter.

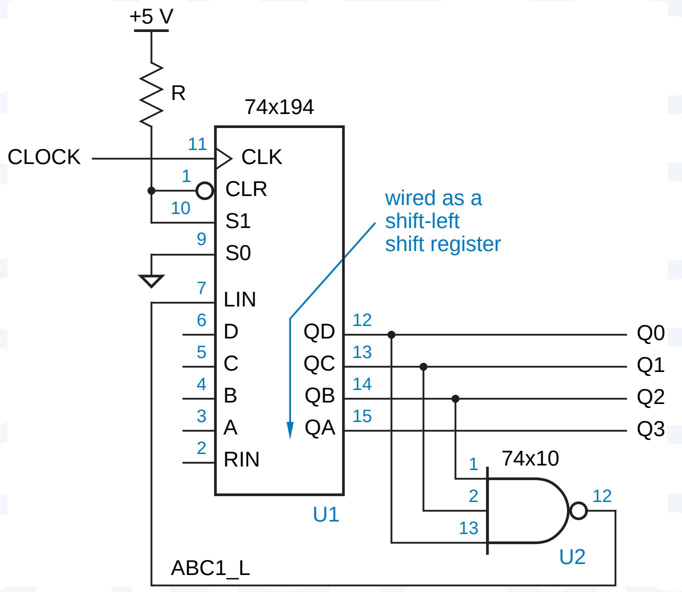


Figure 8-64 Self-correcting four-bit, four-state ring counter with a single circulating 0.

into the normal cycle. Notice that, in this circuit, an explicit RESET signal is not necessarily required. Regardless of the initial state of the shift register on power-up, it reaches state 0001 within four clock ticks. Therefore, an explicit reset signal is required only if it is necessary to ensure that the counter starts up synchronously with other devices in the system or to provide a known starting point in simulation.

For the general case, an n-bit self-correcting ring counter uses an n–1-input NOR gate, and corrects an abnormal state within $n - 1$ clock ticks.

In CMOS and TTL logic families, wide NAND gates are generally easier to come by than NORs, so it may be more convenient to design a self-correcting ring counter as shown in Figure 8-64. States in this counter's normal cycle have a single circulating 0.

The major appeal of a ring counter for control applications is that its states appear in 1-out-of-n decoded form directly on the flip-flop outputs. That is, exactly one flip-flop output is asserted in each state. Furthermore, these outputs are "glitch free"; compare with the binary counter and decoder approach of Figure 8-42 on page 605.

*8.5.7 Johnson Counters

twisted-ring counter
Moebius counter
Johnson counter

An n-bit shift register with the complement of the serial output fed back into the serial input is a counter with $2n$ states and is called a *twisted-ring*, *Moebius*, or *Johnson counter.* Figure 8-65 is the basic circuit for a Johnson counter and Figure 8-66 is its timing diagram. The normal states of this counter are listed in

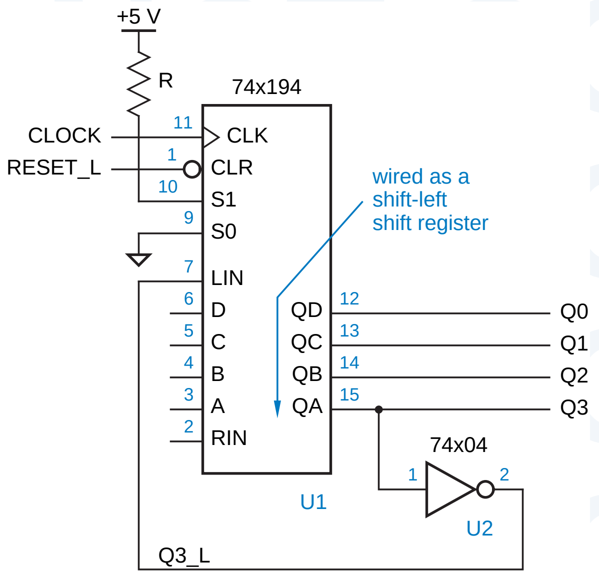


Figure 8-65 Basic four-bit, eight-state Johnson counter.

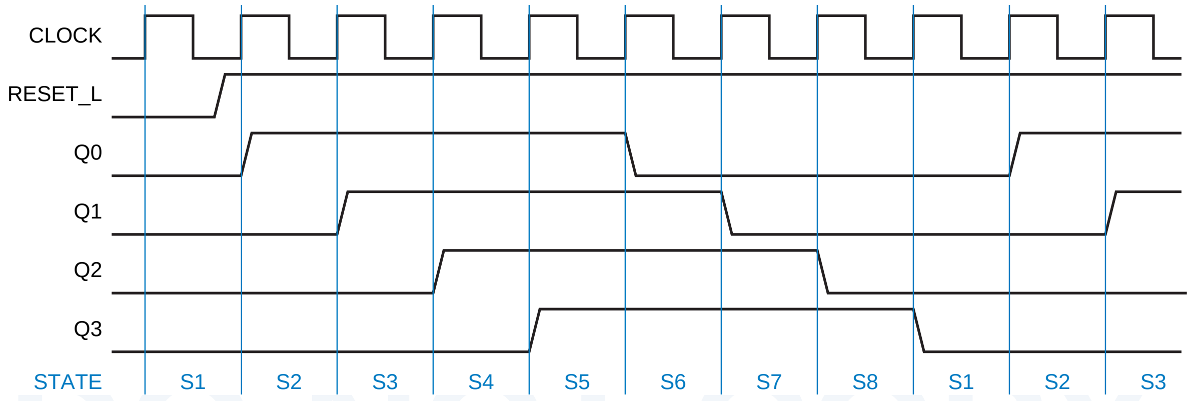


Figure 8-66 Timing diagram for a 4-bit Johnson counter.

Table 8-20 States of a 4-bit Johnson counter.

State Name	*Q3*	*Q2*	*Q1*	*Q0*	*Decoding*
S1	0	0	0	0	Q3′ · Q0′
S2	0	0	0	1	Q1′ · Q0
S3	0	0	1	1	Q2′ · Q1
S4	0	1	1	1	Q3′ · Q2
S5	1	1	1	1	Q3 · Q0
S6	1	1	1	0	Q1 · Q0′
S7	1	1	0	0	Q2 · Q1′
S8	1	0	0	0	Q3 · Q2′

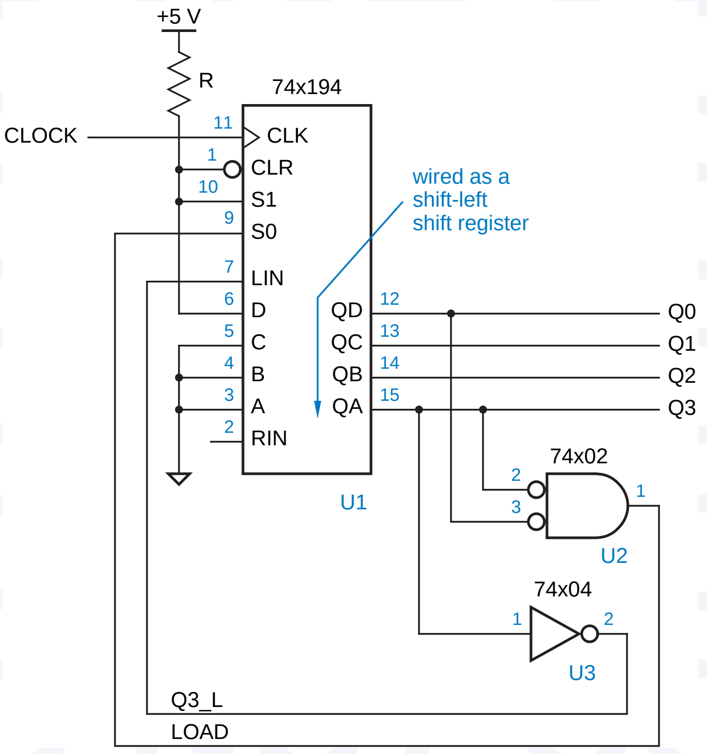


Figure 8-67 Self-correcting four-bit, eight-state Johnson counter.

Table 8-20. If both the true and complemented outputs of each flip-flop are available, each normal state of the counter can be decoded with a 2-input AND or NAND gate, as shown in the table. The decoded outputs are glitch free.

An n-bit Johnson counter has $2^n - 2n$ abnormal states, and is therefore subject to the same robustness problems as a ring counter. A 4-bit *self-correcting Johnson counter* can be designed as shown in Figure 8-67. This circuit loads 0001 as the next state whenever the current state is 0xx0. A similar circuit using a single 2-input NOR gate can perform correction for a Johnson counter with any number of bits. The correction circuit must load 00…01 as the next state whenever the current state is 0x…x0.

self-correcting Johnson counter

THE SELF-CORRECTION CIRCUIT IS ITSELF CORRECT!

We can prove that the Johnson-counter self-correction circuit corrects any abnormal state as follows. An abnormal state can always be written in the form x…x10x…x, since the only states that can't be written in this form are normal states (00…00, 11…11, 01…1, 0…01…1, and 0…01). Therefore, within $n-2$ clock ticks, the shift register will contain 10x…x. One tick later it will contain 0x…x0, and one tick after that the normal state 00…01 will be loaded.

*8.5.8 Linear Feedback Shift Register Counters

The n-bit shift register counters that we've shown so far have far less than the maximum of 2^n normal states. An n-bit *linear feedback shift-register (LFSR) counter* can have $2^n - 1$ states, almost the maximum. Such a counter is often called a *maximum-length sequence generator.*

maximum-length sequence generator

finite fields

The design of LFSR counters is based on the theory of *finite fields*, which was developed by French mathematician Évariste Galois (1811–1832) shortly before he was killed in a duel with a political opponent. The operation of an LFSR counter corresponds to operations in a finite field with 2^n elements.

Figure 8-68 shows the structure of an n-bit LFSR counter. The shift register's serial input is connected to the sum modulo 2 of a certain set of output bits. These feedback connections determine the state sequence of the counter. By convention, outputs are always numbered and shifted in the direction shown.

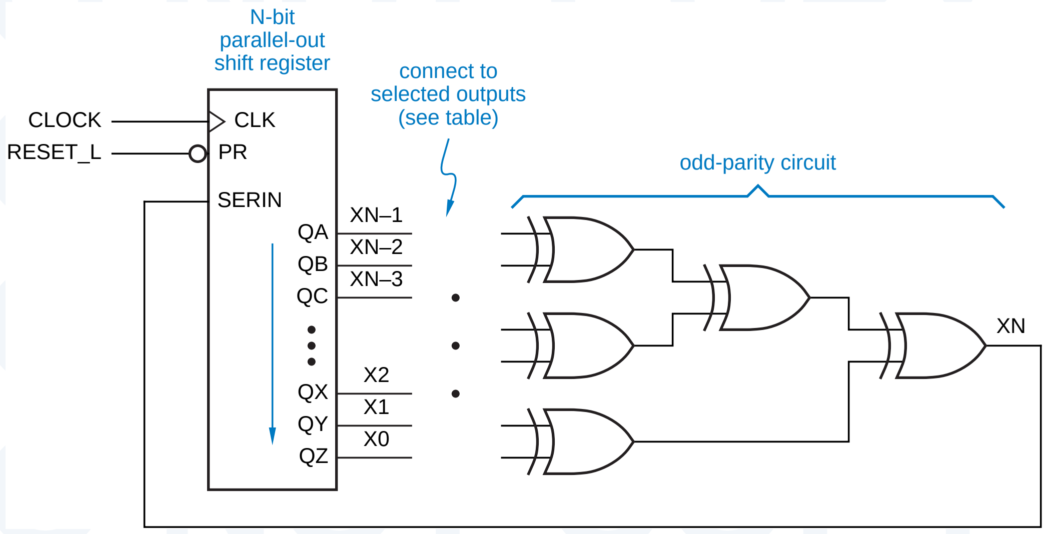
Using finite field theory, it can be shown that for any value of n, there exists at least one feedback equation such that the counter cycles through all $2^n - 1$ nonzero states before repeating. This is called a *maximum-length sequence.*

maximum-length sequence

Table 8-21 lists feedback equations that result in maximum-length sequences for selected values of n. For each value of n greater than 3, there are many other feedback equations that result in maximum-length sequences, all different.

An LFSR counter designed according to Figure 8-68 can never cycle through all 2^n possible states. Regardless of the connection pattern, the next state for the all-0s state is the same—all 0s.

Figure 8-68 General structure of a linear feedback shift-register counter.



n	***Feedback Equation***
2	X2 = X1 ⊕ X0
3	X3 = X1 ⊕ X0
4	X4 = X1 ⊕ X0
5	X5 = X2 ⊕ X0
6	X6 = X1 ⊕ X0
7	X7 = X3 ⊕ X0
8	X8 = X4 ⊕ X3 ⊕ X2 ⊕ X0
12	X12 = X6 ⊕ X4 ⊕ X1 ⊕ X0
16	X16 = X5 ⊕ X4 ⊕ X3 ⊕ X0
20	X20 = X3 ⊕ X0
24	X24 = X7 ⊕ X2 ⊕ X1 ⊕ X0
28	X28 = X3 ⊕ X0
32	X32 = X22 ⊕ X2 ⊕ X1 ⊕ X0

Table 8-21
Feedback equations for linear feedback shift-register counters.

WORKING IN THE FIELD

A finite field has a finite number of elements and two operators, addition and multiplication, that satisfy certain properties. An example of a finite field with P elements, where P is any prime, is the set of integers modulo P. The operators in this field are addition and multiplication modulo P.

According to finite-field theory, if you start with a nonzero element E and repeatedly multiply by a “primitive” element α, after $P-2$ steps you will generate the rest of the field’s nonzero elements in the field before getting back to E. It turns out that in a field with P elements, any integer in the range 2, … , $P-1$ is primitive. You can try this yourself using $P = 7$ and $\alpha = 2$, for example. The elements of the field are 0, 1, … , 6, and the operations are addition and subtraction modulo 7.

The paragraph above gives the basic idea behind maximum-length sequence generators. However, to apply them to a digital circuit, you need a field with 2^n elements, where n is the number of bits required by the application. On one hand, we’re in luck, because Galois proved that there exist finite fields with P^n elements for any integer n, as long as P is prime, including $P = 2$. On the other hand, we’re out of luck, because when $n > 1$, the operators in fields with P^n (including 2^n) elements are quite different from ordinary integer addition and multiplication. Also, primitive elements are harder to find.

If you enjoy math, as I do, you’d probably be fascinated by the finite-field theory that leads to the LFSR circuits for maximum-length sequence generators and other applications; see the References. Otherwise, you can confidently follow the “cookbook” approach in this section.

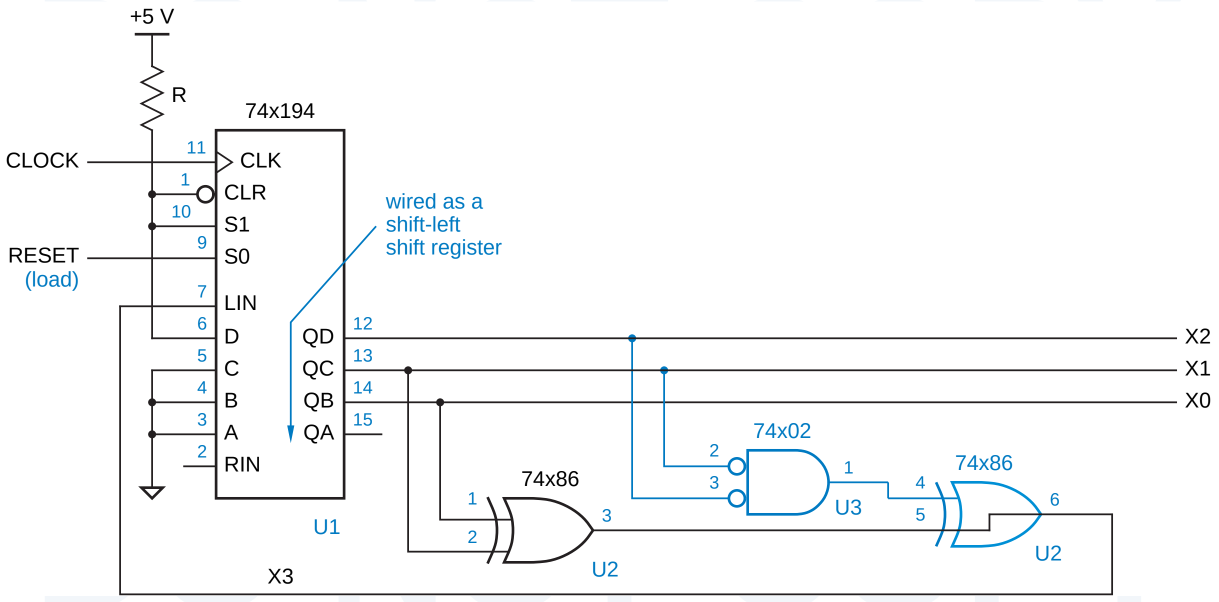


Figure 8-69 A 3-bit LFSR counter; modifications to include the all-0s state are shown in color.

The logic diagram for a 3-bit LFSR counter is shown in Figure 8-69. The state sequence for this counter is shown in the first three columns of Table 8-22. Starting in any nonzero state, 100 in the table, the counter visits seven states before returning to the starting state.

An LFSR counter can be modified to have 2^n states, including the all-0s state, as shown in color for the 3-bit counter in Figure 8-69. The resulting state sequence is given in the last three columns of Table 8-22. In an *n*-bit LFSR

Table 8-22 State sequences for the 3-bit LFSR counter in Figure 8-69.

Original Sequence			*Modified Sequence*		
X2	X1	X0	X2	X1	X0
1	0	0	1	0	0
0	1	0	0	1	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1
0	1	1	0	1	1
0	0	1	0	0	1
1	0	0	0	0	0
.	.	.	1	0	0
.	.	.	.	.	.

counter, an extra XOR gate and an $n - 1$ input NOR gate connected to all shift-register outputs except X0 accomplishes the same thing.

The states of an LFSR counter are not visited in binary counting order. However, LFSR counters are typically used in applications where this characteristic is an advantage. A major application of LFSR counters is in generating test inputs for logic circuits. In most cases, the “pseudo-random” counting sequence of an LFSR counter is more likely than a binary counting sequence to detect errors. LFSRs are also used in the encoding and decoding circuits for certain error-detecting and error-correcting codes, including CRC codes, which we introduced in Section 2.15.4.

In data communications, LFSR counters are often used to “scramble” and “descramble” the data patterns transmitted by high-speed modems and network interfaces, including 100 Mbps Ethernet. This is done by XORing the LFSR’s output with the user data stream. Even when the user data stream contains a long run of 0s or 1s, combining it with the LFSR’s pseudo-random output improves the DC balance of the transmitted signal and creates a rich set of transitions that allows clocking information to be recovered more easily at the receiver.

8.5.9 Shift Registers in ABEL and PLDs

General-purpose shift registers can be specified quite easily in ABEL and fit nicely into typical sequential PLDs. For example, Figure 8-70 and Table 8-23 show how to realize a function similar to that of a 74x194 universal shift register using a 16V8. Notice that one of the I/O pins of the 16V8, pin 12, is used as an input.

The 16V8 realization of the ‘194 differs from the real ’194 in just one way—in the function of the CLR_L input. In the real ’194, CLR_L is an asynchronous input, while in the 16V8 it is sampled along with other inputs at the rising edge of CLK.

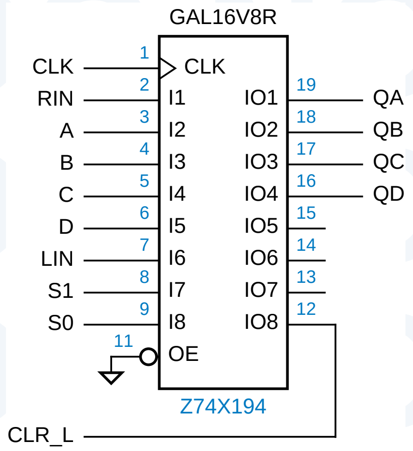


Figure 8-70
PLD realizations of a 74x194-like universal shift register with synchronous clear.

Table 8-23 ABEL program for a 4-bit universal shift register.

```
module Z74x194
title '4-bit Universal Shift Register'
Z74X194 device 'P16V8R';

" Input and output pins
CLK, RIN, A, B, C, D, LIN             pin 1, 2, 3, 4, 5, 6, 7;
S1, S0, CLR_L                         pin 8, 9, 12;
QA, QB, QC, QD                      pin 19, 18, 17, 16 istype 'reg';

" Active-level translation
CLR = !CLR_L;

" Set definitions
INPUT   = [  A,  B,  C, D  ];
LEFTIN  = [ QB, QC, QD, LIN];
RIGHTIN = [RIN, QA, QB, QC ];
OUT     = [ QA, QB, QC, QD ];

CTRL  = [S1,S0];
HOLD  = (CTRL == [0,0]);
RIGHT = (CTRL == [0,1]);
LEFT  = (CTRL == [1,0]);
LOAD  = (CTRL == [1,1]);

equations
OUT.CLK = CLK;

OUT := !CLR & (
           HOLD & OUT
         # RIGHT & RIGHTIN
         # LEFT & LEFTIN
         # LOAD & INPUT);

end Z74x194
```

If you really need to provide an asynchronous clear input, you can use the 22V10, which provides a single product line to control the reset inputs of all of its flip-flops. This requires only a few changes in the original program (see Exercise 8.51).

The flexibility of ABEL can be used to create shift registers circuits with more or different functionality. For example, Table 8-24 defines an 8-bit shift register that can be cleared, loaded with a single 1 in any bit position, shifted left, shifted right, or held. The operation to be performed at each clock tick is specified by a 4-bit operation code, OP[3:0]. Despite the large number of "WHEN" cases, the circuit can be synthesized with only five product terms per output.

Table 8-24 ABEL program for a multi-function shift register.

```
module shifty
title '8-bit shift register with decoded load'

" Inputs and Outputs
CLK, OP3..OP0               pin;
Q7..Q0                      pin istype 'reg';

" Definitions

Q = [Q7..Q0];
OP = [OP3..OP0];

HOLD = (OP == 0);
CLEAR = (OP == 1);
LEFT = (OP == 2);
RIGHT = (OP == 3);
NOP = (OP >= 4) & (OP < 8);
LOADQ0 = (OP == 8);
LOADQ1 = (OP == 9);
LOADQ2 = (OP == 10);
LOADQ3 = (OP == 11);
LOADQ4 = (OP == 12);
LOADQ5 = (OP == 13);
LOADQ6 = (OP == 14);
LOADQ7 = (OP == 15);

Equations

Q.CLK = CLK;

WHEN HOLD THEN Q := Q;
ELSE WHEN CLEAR THEN Q := 0;
ELSE WHEN LEFT THEN Q := [Q6..Q0, Q7];
ELSE WHEN RIGHT THEN Q := [Q0, Q7..Q1];
ELSE WHEN LOADQ0 THEN Q := 1;
ELSE WHEN LOADQ1 THEN Q := 2;
ELSE WHEN LOADQ2 THEN Q := 4;
ELSE WHEN LOADQ3 THEN Q := 8;
ELSE WHEN LOADQ4 THEN Q := 16;
ELSE WHEN LOADQ5 THEN Q := 32;
ELSE WHEN LOADQ6 THEN Q := 64;
ELSE WHEN LOADQ7 THEN Q := 128;
ELSE Q := Q;

end shifty
```

Table 8-25 Program for an 8-bit ring counter.

```
module Ring8
title '8-bit Ring Counter'

" Inputs and Outputs
MCLK, CNTEN, RESTART                pin;
S0..S7                              pin istype 'reg';

equations

[S0..S7].CLK = MCLK;

S0 := CNTEN & !S0 & !S1 & !S2 & !S3 & !S4 & !S5 & !S6  " Self-sync
    # !CNTEN & S0                                       " Hold
    # RESTART;                                          " Start with one 1
[S1..S7] := !RESTART & ( !CNTEN & [S1..S7]      " Shift
                       # CNTEN & [S0..S6] );   " Hold
end Ring8
```

ABEL can be used readily to specify shift register counters of the various types that we introduced in previous subsections. For example, Table 8-25 is the program for an 8-bit ring counter. We've used the extra capability of the PLD to add two functions not present in our previous MSI designs: counting occurs only if CNTEN is asserted, and the next state is forced to S0 if RESTART is asserted.

Ring counters are often used to generate multiphase clocks or enable signals in digital systems, and the requirements in different systems are many and varied. The ability to reprogram the counter's behavior easily is a distinct advantage of an HDL-based design.

Figure 8-71 shows a set of clock or enable signals that might be required in a digital system with six distinct phases of operation. Each phase lasts for two ticks of a master clock signal, MCLK, during which the corresponding active-low phase-enable signal Pi_L is asserted. We can obtain this sort of timing from a ring counter if we provide an extra flip-flop to count the two ticks of each phase, so that a shift occurs on the *second* tick of each phase.

The timing generator can be built with a few inputs and outputs of a PLD. Three control inputs are provided, with the following behavior:

RESET	When this input is asserted, no outputs are asserted. The counter always goes to the first tick of phase 1 after RESET is negated.
RUN	When asserted, this input allows the counter to advance to the second tick of the current phase, or to the first tick of the next phase; otherwise, the current tick of the current phase is extended.
RESTART	Asserting this input causes the counter to go back to the first tick of phase 1, even if RUN is not asserted.

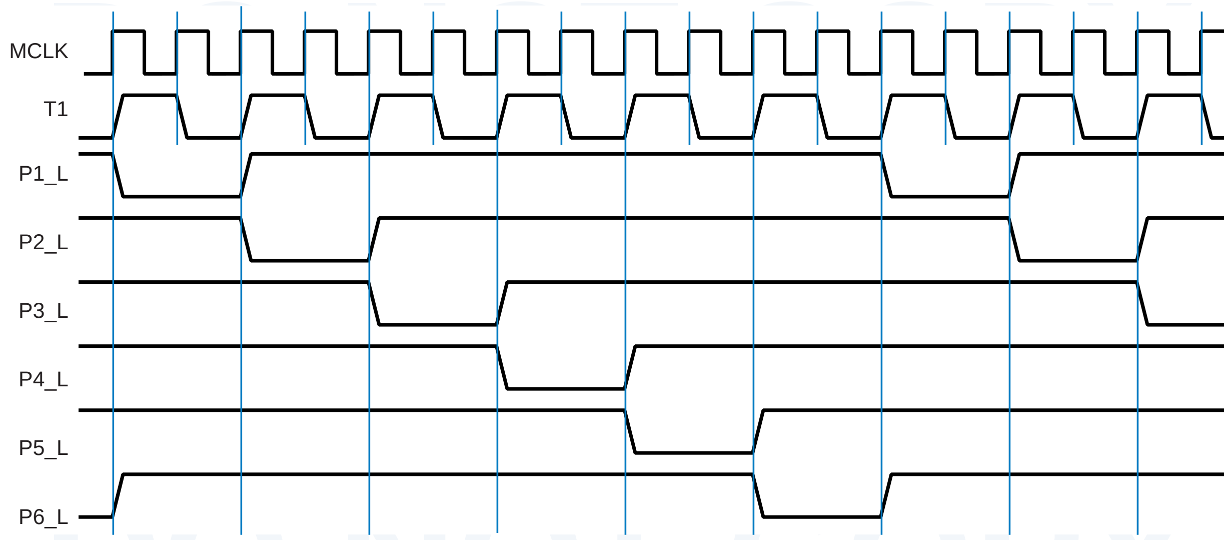


Figure 8-71 Six-phase timing waveforms required in a certain digital system.

Table 8-26 Program for a six-phase waveform generator.

```
module TIMEGEN6
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                  pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L     pin istype 'reg';

" State definitions
PHASES = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
NEXTPH = [P6_L, P1_L, P2_L, P3_L, P4_L, P5_L];
SRESET = [1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1];

equations
T1.CLK = MCLK; PHASES.CLK = MCLK;

WHEN RESET THEN {T1 := 1; PHASES := SRESET;}
ELSE WHEN (PHASES==SRESET) # RESTART THEN {T1 := 1; PHASES := P1;}
ELSE WHEN RUN & T1 THEN {T1 := 0; PHASES := PHASES;}
ELSE WHEN RUN & !T1 THEN {T1 := 1; PHASES := NEXTPH;}
ELSE {T1 := T1; PHASES := PHASES;}

end TIMEGEN6
```

Table 8-26 is a program that creates the required behavior. Notice the use of sets to specify the ring counter's behavior very concisely, with the RESET, RESTART, and RUN having the specified behavior in any counter phase.

Table 8-27 Alternate program for the waveform generator.

```
module TIMEGN6A
title 'Six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                 pin;
T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L    pin istype 'reg';

" State definitions
TSTATE = [T1, P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];
SRESET = [1, 1, 1, 1, 1, 1, 1];
P1F =    [1, 0, 1, 1, 1, 1, 1];
P1S =    [0, 0, 1, 1, 1, 1, 1];
P2F =    [1, 1, 0, 1, 1, 1, 1];
P2S =    [0, 1, 0, 1, 1, 1, 1];
P3F =    [1, 1, 1, 0, 1, 1, 1];
P3S =    [0, 1, 1, 0, 1, 1, 1];
P4F =    [1, 1, 1, 1, 0, 1, 1];
P4S =    [0, 1, 1, 1, 0, 1, 1];
P5F =    [1, 1, 1, 1, 1, 0, 1];
P5S =    [0, 1, 1, 1, 1, 0, 1];
P6F =    [1, 1, 1, 1, 1, 1, 0];
P6S =    [0, 1, 1, 1, 1, 1, 0];

equations
TSTATE.CLK = MCLK;
WHEN RESET THEN TSTATE := SRESET;

state_diagram TSTATE

state SRESET: IF RESET THEN SRESET ELSE P1F;

state P1F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1S ELSE P1F;

state P1S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2F ELSE P1S;

state P2F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P2S ELSE P2F;

state P2S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3F ELSE P2S;

state P3F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P3S ELSE P3F;

state P3S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4F ELSE P3S;

state P4F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P4S ELSE P4F;

state P4S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5F ELSE P4S;
```

Table 8-27 (continued) Alternate program for the waveform generator.

```
state P5F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P5S ELSE P5F;

state P5S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6F ELSE P5S;

state P6F: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P6S ELSE P6F;

state P6S: IF RESET THEN SRESET ELSE IF RESTART THEN P1F
           ELSE IF RUN THEN P1F ELSE P6S;

end TIMEGN6A
```

The same timing-generator behavior in Figure 8-71 can be specified using a state-machine design approach, as shown in Table 8-27. This ABEL program, though longer, generates the same external behavior during normal operation as the previous one, and from a certain point of view it may be easier to understand. However, its realization requires 8 to 20 AND terms per output, compared to only 3 to 5 per output for the original ring-counter version. This is a good example of how you can improve circuit efficiency and performance by adapting a standard, simple structure to a "custom" design problem, rather than grinding out a brute-force state machine.

RELIABLE RESET

Notice in Table 8-27 that TSTATE is assigned a value in the equations section of the program, as well as being used in the state_diagram section. This was done very a very specific purpose, to ensure that the program goes to the SRESET state from any undefined state, as explained below.

ABEL augments the on-set for an output each time the output appears on the left-hand side of an equation, as we explained for combinational outputs on page 252. In the case of registered outputs, ABEL also augments the on-set of each state variable in the state vector for each "state" definition in a state_diagram. For each state-variable output, all of the input combinations that cause that output to be 1 in each state are added to the output's on-set.

The state machine in Table 8-27 has 2^7 or 128 states in total, of which only 13 are explicitly defined and have a transition into SRESET. Nevertheless, the WHEN equation ensures that anytime that RESET is asserted, the machine goes to the SRESET state. This is true regardless of the state definitions in the state_diagram. When RESET is asserted, the all-1s state encoding of SRESET is, in effect, ORed with the next state, if any, specified by the state_diagram. This approach to reliable reset would not be possible if SRESET were encoded as all 0s, for example.

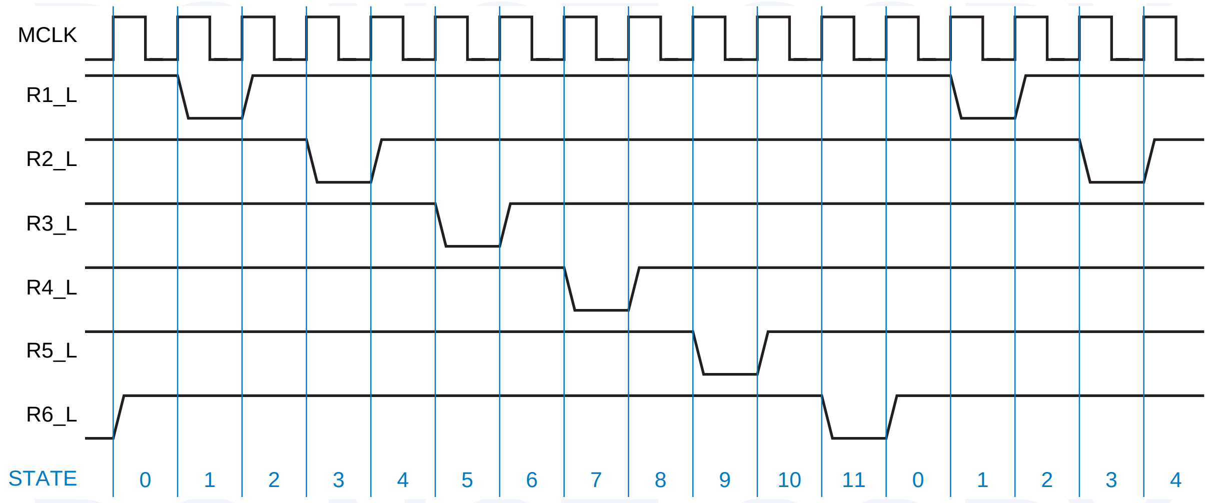


Figure 8-72 Modified timing waveforms for a digital system.

Now let's look at a variation of the previous timing waveforms that might be required in a different system. Figure 8-72 is similar to the previous waveforms, except that each phase output Ri is asserted for only one clock tick per phase. This change has a subtle but important effect on the design approach.

In the original design, we used a six-bit ring counter and one auxiliary state bit T1 to keep track of the two states within each phase. With the new waveforms, this is not possible. In the states between active-low pulses (STATE = 0, 2, 4, etc. in Figure 8-72), the phase outputs are all negated, so they can no longer be used to figure out which state should be visited next. Something else is needed to keep track of the state.

There are many different ways to solve this problem. One idea is to start with the original design in Table 8-26, but use the phase outputs P1_L, P2_L,

Table 8-28 Additions to Table 8-26 for a modified six-phase waveform generator.

```
module TIMEG12K
...
R1_L, R2_L, R3_L, R4_L, R5_L, R6_L         pin istype 'com';
...
OUTPUTS = [R1_L, R2_L, R3_L, R4_L, R5_L, R6_L];

equations
...
!OUTPUTS = !PHASES & !T1;

end TIMEG12K
```

and so on as internal states only. Then, each phase output Ri_L can be defined as a Moore-type combinational output that is asserted when the corresponding Pi_L is asserted *and* we are in the second tick of a phase. The additional ABEL code to support this first approach is shown in Table 8-28.

This first approach is easy, and it works just fine if the Pi_L signals are going to be used only as enables or other control inputs. However, it's a bad idea if these signals are going to be used as clocks, because they may have glitches, as we'll now explain. The Pi_L and T1 signals are all outputs from flip-flops clocked by the same master clock MCLK. Although these signals change at approximately the same time, their timing is never quite exact. One output may change sooner than another; this is called *output timing skew*. For example, suppose that on the transition from state 1 to 2 in Figure 8-71, P2_L goes LOW before T1 goes HIGH. In this case, a short glitch could appear on the R2_L output.

output timing skew

To get glitch-free outputs, we should design the circuit so that each phase output is the a registered output. One way to do this is to build a 12-bit ring counter, and only use alternate outputs to yield the desired waveforms; an ABEL program using this approach is shown in Table 8-29.

Table 8-29 ABEL program for a modified six-phase waveform generator.

```
module TIMEG12
title 'Modified six-phase Master Timing Generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART                 pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L        pin istype 'reg';
P1A, P2A, P3A, P4A, P5A, P6A              pin istype 'reg';

" State definitions
PHASES = [P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A, P6_L];
NEXTPH = [P6_L, P1A, P1_L, P2A, P2_L, P3A, P3_L, P4A, P4_L, P5A, P5_L, P6A];
SRESET = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
P1 =     [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];

equations

PHASES.CLK = MCLK;

WHEN RESET THEN PHASES := SRESET;
ELSE WHEN RESTART # (PHASES == SRESET) THEN PHASES := P1;
ELSE WHEN RUN THEN PHASES := NEXTPH;
ELSE PHASES := PHASES;

end TIMEG12
```

Table 8-30 Counter-based program for six-phase waveform generator.

```
module TIMEG12A
title 'Counter-based six-phase master timing generator'

" Input and Output pins
MCLK, RESET, RUN, RESTART              pin;
P1_L, P2_L, P3_L, P4_L, P5_L, P6_L     pin istype 'reg';
CNT3..CNT0                             pin istype 'reg';

" Definitions
CNT = [CNT3..CNT0];
P_L = [P1_L, P2_L, P3_L, P4_L, P5_L, P6_L];

equations

CNT.CLK = MCLK;  P_L.CLK = MCLK;

WHEN RESET THEN CNT := 15
ELSE WHEN RESTART THEN CNT := 0
ELSE WHEN (RUN & (CNT < 11)) THEN CNT := CNT + 1
ELSE WHEN RUN THEN CNT := 0
ELSE CNT := CNT;

P1_L := !(CNT == 0);
P2_L := !(CNT == 2);
P3_L := !(CNT == 4);
P4_L := !(CNT == 6);
P5_L := !(CNT == 8);
P6_L := !(CNT == 10);

end TIMEG12A
```

Still another approach is to recognize that since the waveforms cycle through 12 states, we can build a modulo-12 binary counter and decode the states of that counter. An ABEL program using this approach is shown in Table 8-30. The states of the counter correspond to the “STATE” values shown in Figure 8-72. Since the phase outputs are registered, they are glitch-free. Note that they are decoded one cycle early, to account for the one-tick decoding delay. Also, during reset, the counter is forced to state 15 rather than 0, so that the P1_L output is not asserted during reset.

8.5.10 Shift Registers in VHDL

Shift registers can be specified structurally or behaviorally in VHDL; we’ll look at a few behavioral descriptions and applications. Table 8-31 is the function table for an 8-bit shift register with an extended set of functions. In addition to the hold, load, and shift functions of the 74x194 and 74x299, it performs circular

Table 8-31 Function table for an extended-function 8-bit shift register.

	Inputs			Next state							
Function	**S2**	**S1**	**S0**	**Q7***	**Q6***	**Q5***	**Q4***	**Q3***	**Q2***	**Q1***	**Q0***
Hold	0	0	0	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
Load	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0
Shift right	0	1	0	RIN	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift left	0	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIN
Shift circular right	1	0	0	Q0	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift circular left	1	0	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	Q7
Shift arithmetic right	1	1	0	Q7	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift arithmetic left	1	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0

and arithmetic shift operations. In the *circular shift* operations, the bit that "falls off" one end during a shift is fed back into the other end. In the *arithmetic shift* operations, the edge input is set up for multiplication or division by 2; for a left shift, the right input is 0, and for a right shift, the leftmost (sign) bit is replicated.

circular shift
arithmetic shift

A behavioral VHDL program for the extended-function shift register is shown in Table 8-32. As in previous examples, we define a process and use the `event` attribute on the `CLK` signal to obtain the desired edge-triggered behavior. Several other features of this program are worth noting:

- An internal signal, `IQ`, is used for what eventually becomes the `Q` output, so it can be both read and written by process statements. Alternatively, we could have defined the `Q` output as type "`buffer`".
- The `CLR` input is asynchronous; because it's in the process sensitivity list, it is tested whenever it changes. And the `IF` statement is structured so that `CLR` takes precedence over any other condition.
- A `CASE` statement is used to define the operation of the shift register for the eight possible values of the select inputs `S(2 downto 0)`.
- In the `CASE` statement, the "`when others`" case is required to prevent the compiler from complaining about approximately 2^{32} uncovered cases!
- The "`null`" statement indicates that no action is taken in certain cases. In case 1, note that no action is required; the default is for a signal to hold its value unless otherwise stated.
- In most of the cases, the concatenation operator "`&`" is used to construct an 8-bit array from a 7-bit subset of `IQ` and one other bit.

Table 8-32 VHDL program for an extended-function 8-bit shift register.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity Vshftreg is
    port (
        CLK, CLR, RIN, LIN: in STD_LOGIC;
        S: in STD_LOGIC_VECTOR (2 downto 0); -- function select
        D: in STD_LOGIC_VECTOR (7 downto 0); -- data in
        Q: out STD_LOGIC_VECTOR (7 downto 0) -- data out
    );
end Vshftreg;

architecture Vshftreg_arch of Vshftreg is
signal IQ: STD_LOGIC_VECTOR (7 downto 0);
begin
process (CLK, CLR, IQ)
  begin
    if (CLR='1') then IQ <= (others=>'0'); -- Asynchronous clear
    elsif (CLK'event and CLK='1') then
      case CONV_INTEGER(S) is
        when 0 => null;                        -- Hold
        when 1 => IQ <= D;                     -- Load
        when 2 => IQ <= RIN & IQ(7 downto 1);  -- Shift right
        when 3 => IQ <= IQ(6 downto 0) & LIN;  -- Shift left
        when 4 => IQ <= IQ(0) & IQ(7 downto 1); -- Shift circular right
        when 5 => IQ <= IQ(6 downto 0) & IQ(7); -- Shift circular left
        when 6 => IQ <= IQ(7) & IQ(7 downto 1); -- Shift arithmetic right
        when 7 => IQ <= IQ(6 downto 0) & '0';  -- Shift arithmetic left
        when others => null;
      end case;
    end if;
    Q <= IQ;
  end process;
end Vshftreg_arch;
```

- Because of VHDL's strong requirements for type matching, we used the CONV_INTEGER function in the IEEE.std_logic_unsigned library to convert the STD_LOGIC_VECTOR select input S to an integer in the CASE statement. Alternatively, we could have written each case label as a STD_LOGIC_VECTOR (e.g., ('0','1','1') instead of integer 3).

One application of shift registers is in ring counters, as in the six-phase waveform generator that we described on page 638 with the waveforms in Figure 8-71. A VHDL program that provides the corresponding behavior is

Table 8-33 VHDL program for a six-phase waveform generator.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimegn6 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimegn6;

architecture Vtimegn6_arch of Vtimegn6 is
signal IP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
signal T1: STD_LOGIC;                 -- first tick within phase
begin
process (MCLK, IP)
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then
        T1 <= '1'; IP <= ('0','0','0','0','0','0');
      elsif ((IP=('0','0','0','0','0','0')) or (RESTART='1')) then
        T1 <= '1'; IP <= ('1','0','0','0','0','0');
      elsif (RUN='1') then
        T1 <= not T1;
        if (T1='0') then IP <= IP(6) & IP(1 to 5); end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimegn6_arch;
```

shown in Table 8-33. As in the previous VHDL example, an internal active-high signal vector, IP, is used for reading and writing what eventually becomes the circuit's output; this internal signal is conveniently inverted in the last statement to obtain the required active-low output signal vector. The rest of the program is straightforward, but notice that it has three levels of nested IF statements.

A possible modification to the preceding application is to produce output waveforms that are asserted only during the second tick of each two-tick phase; such waveforms were shown in Figure 8-72 on page 642. One way to do this is to create a 12-bit ring counter, and use only alternate outputs. In the VHDL realization, only the six phase outputs, P_L(1 to 6), would appear in the entity definition. The additional six signals, which we name NEXTP(1 to 6), are local to the architecture definition. Figure 8-73 shows the relationship of these signals for shift-register operation, and Table 8-34 is the VHDL program.

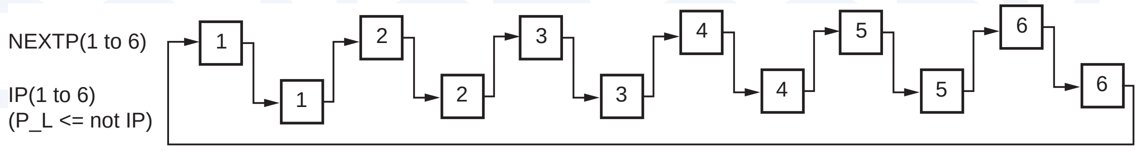


Figure 8-73 Shifting sequence for waveform generator 12-bit ring counter.

As in the previous program, a 6-bit active-high signal, IP, is declared in the architecture body and used for reading and writing what eventually becomes the circuit's active-low output, P_L. The additional 6-bit signal, NEXTP, holds the remaining six bits of state. Constants IDLE and FIRSTP are used to improve the program's readability.

Table 8-34 VHDL program for a modified six-phase waveform generator.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vtimeg12 is
    port (
        MCLK, RESET, RUN, RESTART: in STD_LOGIC; -- clock, control inputs
        P_L: out STD_LOGIC_VECTOR (1 to 6)       -- active-low phase outputs
    );
end Vtimeg12;

architecture Vtimeg12_arch of Vtimeg12 is
signal IP, NEXTP: STD_LOGIC_VECTOR (1 to 6); -- internal active-high phase signals
begin
process (MCLK, IP, NEXTP)
  variable TEMP: STD_LOGIC_VECTOR (1 to 6);  -- temporary for signal shift
  constant IDLE: STD_LOGIC_VECTOR (1 to 6) := ('0','0','0','0','0','0');
  constant FIRSTP: STD_LOGIC_VECTOR (1 to 6) := ('1','0','0','0','0','0');
  begin
    if (MCLK'event and MCLK='1') then
      if (RESET='1') then IP <= IDLE;  NEXTP <= IDLE;
      elsif (RESTART='1') or (IP=IDLE and NEXTP=IDLE) then IP <= IDLE;  NEXTP <= FIRSTP;
      elsif (RUN='1') then
        if (IP=IDLE) and (NEXTP=IDLE) then NEXTP <= FIRSTP;
        else TEMP := IP;  IP <= NEXTP;  NEXTP <= TEMP(6) & TEMP(1 to 5);
        end if;
      end if;
    end if;
    P_L <= not IP;
  end process;
end Vtimeg12_arch;
```

Notice that a six-bit variable, `TEMP`, is used just as a temporary place to hold the old value of `IP` when shifting occurs—`IP` is loaded with `NEXTP`, and `NEXTP` is loaded with the shifted, old value of `IP`. Because the assignment statements in a process are executed *sequentially*, we couldn't get away with just writing "`IP <= NEXTP; NEXTP <= IP(6) & IP(1 to 5);`". If we did that, then `NEXTP` would pick up the *new* value of `IP`, not the old. Notice also that since `TEMP` is a variable, not a signal, its value is not preserved between process invocations. Thus, the VHDL compiler does not synthesize any flip-flops to hold `TEMP`'s value.

*8.6 Iterative versus Sequential Circuits

We introduced iterative circuits in Section 5.9.2. The function of an *n*-module iterative circuit can be performed by a sequential circuit that uses just one copy of the module but requires *n* steps (clock ticks) to obtain the result. This is an excellent example of a space/time trade-off in digital design.

As shown in Figure 8-74, flip-flops are used in the sequential-circuit version to store the cascading outputs at the end of each step; the flip-flop outputs are used as the cascading inputs at the beginning of the next step. The flip-flops must be initialized to the boundary-input values before the first clock tick, and they contain the boundary-output values after the *n*th tick.

Since an iterative circuit is a combinational circuit, all of its primary and boundary inputs may be applied simultaneously, and its primary and boundary outputs are all available after a combinational delay. In the sequential-circuit version, the primary inputs must be delivered sequentially, one per clock tick, and the primary outputs are produced with similar timing. Therefore, serial-out shift registers are often used to provide the inputs, and serial-in shift registers are used to collect the outputs. For this reason, the sequential-circuit version of an "iterative widget" is often called a "serial widget."

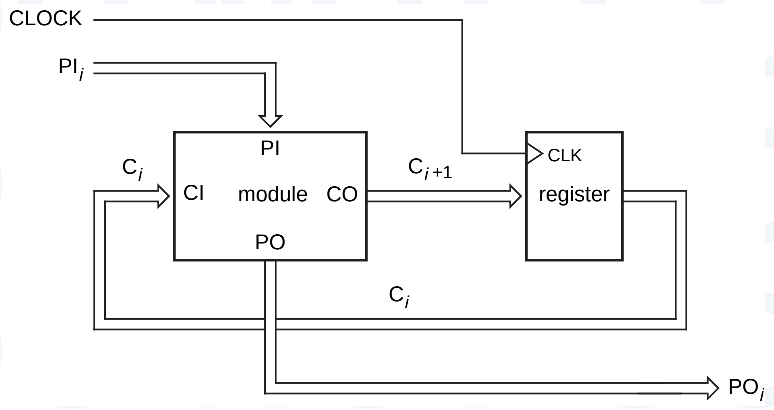


Figure 8-74 General structure of the sequential-circuit version of an iterative circuit.

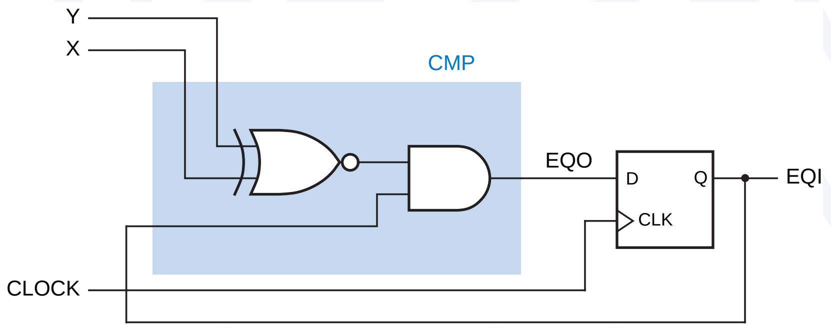


Figure 8-75 Simplified serial comparator circuit.

serial comparator

For example, Figure 8-75 shows the basic design for a *serial comparator* circuit. The shaded block is identical to the module used in the iterative comparator of Figure 5-80 on page 385. The circuit is drawn in more detail using SSI chips in Figure 8-76. In addition, we have provided a synchronous reset input that, when asserted, forces the initial value of the cascading flip-flop to 1 at the next clock tick. The initial value of the cascading flip-flop corresponds to the boundary input in the iterative comparator.

With the serial comparator, an n-bit comparison requires $n + 1$ clock ticks. RESET_L is asserted at the first clock tick. RESET_L is negated and data bits are applied at the next n ticks. The EQI output gives the comparison result during the clock period following the last tick. A timing diagram for two successive 4-bit comparisons is shown in Figure 8-77. The spikes in the EQO waveform indicate the time when the combinational outputs are settling in response to new X and Y input values.

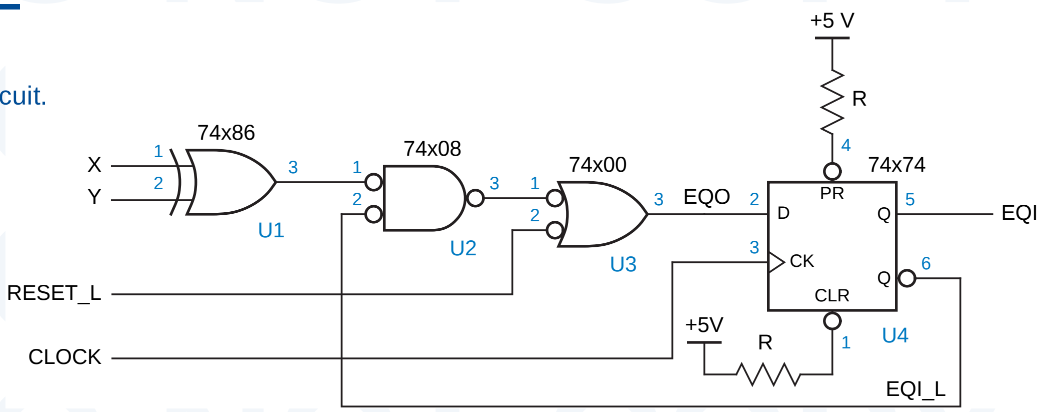


Figure 8-76 Detailed serial comparator circuit.

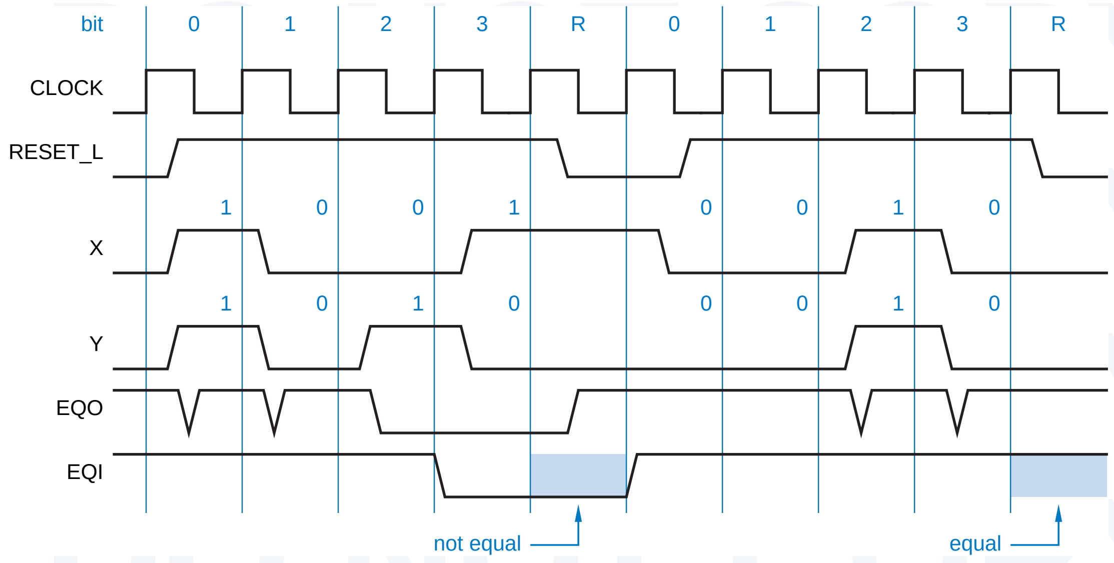


Figure 8-77 Timing diagram for serial comparator circuit.

A *serial binary adder* circuit for addends of any length can be constructed from a full adder and a D flip-flop, as shown in Figure 8-78. The flip-flop, which stores the carry between successive bits of the addition, is cleared to 0 at reset. Addend bits are presented serially on the A and B inputs, starting with the LSB, and sum bits appear on S in the same order.

serial binary adder

Because of the large size and high cost of digital logic circuits in the early days, many computers and calculators used serial adders and other serial versions of iterative circuits to perform arithmetic operations. Even though these arithmetic circuits aren't used much today, they are an instructive reminder of the space/time trade-offs that are possible in digital design.

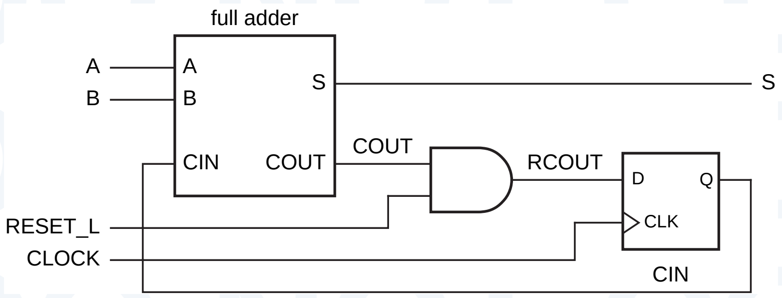


Figure 8-78 Serial binary adder circuit.

8.7 Synchronous Design Methodology

synchronous system

In a *synchronous system*, all flip-flops are clocked by the same, common clock signal, and preset and clear inputs are not used, except for system initialization. Although it's true that all the world does not march to the tick of a common clock, within the confines of a digital system or subsystem we can make it so. When we interconnect digital systems or subsystems that use different clocks, we can usually identify a limited number of asynchronous signals that need special treatment, as we'll show later, in Section 8.8.3.

Races and hazards are not a problem in synchronous systems, for two reasons. First, the only fundamental-mode circuits that might be subject to races or essential hazards are predesigned elements, such as discrete flip-flops or ASIC cells, that are guaranteed by the manufacturer to work properly. Second, even though the combinational circuits that drive flip-flop control inputs may contain static or dynamic or function hazards, these hazards have no effect, since the control inputs are sampled only *after* the hazard-induced glitches have had a chance to settle out.

Aside from designing the functional behavior of each state machine, the designer of a practical synchronous system or subsystem must perform just three well-defined tasks to ensure reliable system operation:

1. Minimize and determine the amount of clock skew in the system, as discussed in Section 8.8.1.
2. Ensure that flip-flops have positive setup- and hold-time margins, including an allowance for clock skew, as described in Section 8.1.4.
3. Identify asynchronous inputs, synchronize them with the clock, and ensure that the synchronizers have an adequately low probability of failure, as described in Sections 8.8.3 and 8.9.

Before we get into these issues, in this section we'll look at a general model for synchronous system structure and an example.

8.7.1 Synchronous System Structure

The sequential-circuit design examples that we gave in Chapter 7 were mostly individual state machines with a small number of states. If a sequential circuit has more than a few flip-flops, then it's not desirable (and often not possible) to treat the circuit as a single, monolithic state machine, because the number of states would be too large to handle.

Fortunately, most digital systems or subsystems can be partitioned into two or more parts. Whether the system processes numbers, digitized voice signals, or a stream of spark-plug pulses, a certain part of the system, which we'll call the *data unit*, can be viewed as storing, routing, combining, and generally processing "data." Another part, which we'll call the *control unit*, can be viewed as starting and stopping actions in the data unit, testing conditions, and deciding

data unit
control unit

what to do next according to circumstances. In general, only the control unit must be designed as a state machine. The data unit and its components are typically handled at a higher level of abstraction, such as:

- *Registers*. A collection of flip-flops is loaded in parallel with many bits of "data," which can then be used or retrieved together.
- *Specialized functions*. These include multibit counters and shift registers, which increment or shift their contents on command.
- *Read/write memory*. Individual latches or flip-flops in a collection of the same can be written or read out.

The first two topics above were discussed earlier in this chapter, and the last is discussed in Chapter 11.

Figure 8-79 is a general block diagram of a system with a control unit and a data unit. We have also included explicit blocks for input and output, but we could have just as easily absorbed these functions into the data unit itself. The control unit is a state machine whose inputs include *command inputs* that indicate how the machine is to function, and *condition inputs* provided by the data unit. The command inputs may be supplied by another subsystem or by a user to set the general operating mode of the control state machine (RUN/HALT, NORMAL/TURBO, etc.), while the condition inputs allow the control state machine unit to change its behavior as required by circumstances in the data unit (ZERO_DETECT, MEMORY_FULL, etc.).

command input
condition input

A key characteristic of the structure in Figure 8-79 is that the control, data, input, and output units all use the same common clock. Figure 8-80 illustrates the operations of the control and data units during a typical clock cycle:

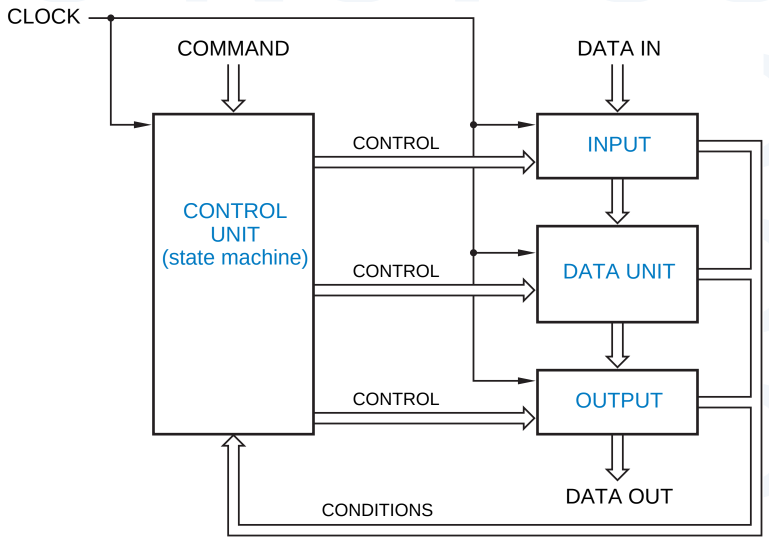


Figure 8-79 Synchronous system structure.

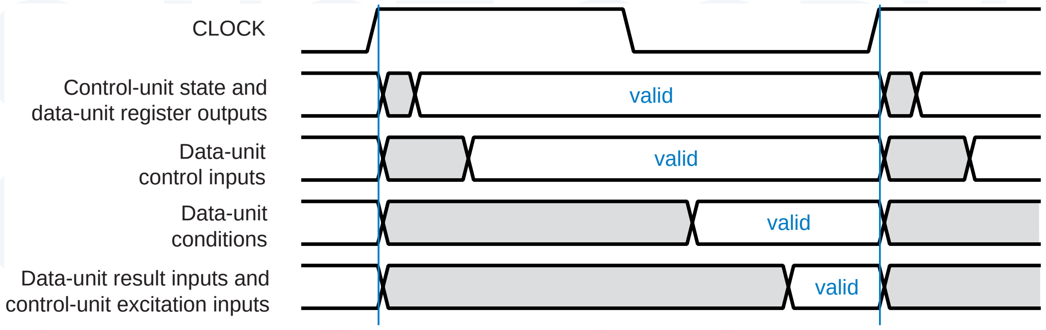

CLOCK
Control-unit state and data-unit register outputs
valid
Data-unit control inputs
valid
Data-unit conditions
valid
Data-unit result inputs and control-unit excitation inputs
valid

Figure 8-80 Operations during one clock cycle in a synchronous system.

1. Shortly after the beginning of the clock period, the control-unit state and the data-unit register outputs are valid.
2. Next, after a combinational logic delay, Moore-type outputs of the control-unit state machine become valid. These signals are control *inputs* to the data unit. They determine what data-unit functions are performed in the rest of the clock period, for example, selecting memory addresses, multiplexer paths, and arithmetic operations.
3. Near the end of the clock period, data-unit condition outputs such as zero- or overflow-detect are valid, and are made available to the control unit.
4. At the end of the clock period, just before the setup-time window begins, the next-state logic of the control-unit state machine has determined the next state based on the current state and command and condition inputs. At about the same time, computational results in the data unit are available to be loaded into data-unit registers.
5. After the clock edge, the whole cycle may repeat.

Data-unit control inputs, which are control-unit state-machine outputs, may be of the Moore, Mealy, or pipelined Mealy type; timing for the Moore type was shown in Figure 8-80. Moore-type and pipelined-Mealy-type outputs control the data unit's actions strictly according to the current state and past inputs, which do not depend on *current* conditions in the data unit. In contrast, Mealy-type outputs may select different actions in the data unit according to *current* conditions in the data unit. This increases flexibility, but typically also

PIPELINED MEALY OUTPUTS

Some state machines have pipelined Mealy outputs, discussed in Section 7.3.2. In Figure 8-80, pipelined Mealy outputs would typically be valid early in the cycle, at the same time as control-unit state outputs. Early validity of these outputs, compared to Moore outputs that must go through a combinational logic delay, may allow the entire system to operate at a faster clock rate.

increases the minimum clock period for correct system operation, since the delay path may be much longer. Also, Mealy-type outputs must not create feedback loops. For example, a signal that adds 1 to an adder's input if the adder output is nonzero causes an oscillation if the adder output is –1.

8.7.2 A Synchronous System Design Example

To give you an overview of several elements of synchronous system design, this subsection presents a representative example of a synchronous system. The example is a *shift-and-add multiplier* for unsigned integers using the algorithm of Section 2.8. Its data unit uses standard combinational and sequential building blocks, and its control unit is described by a state diagram

shift-and-add multiplier

Figure 8-81 illustrates data-unit registers and functions that are used to perform an 8-bit multiplication:

MPY/LPROD	A shift register that initially stores the multiplier, and accumulates the low-order bits of the product as the algorithm is executed.
HPROD	A register that is initially cleared, and accumulates the high-order bits of the product as the algorithm is executed.
MCND	A register that stores the multiplicand throughout the algorithm.
F	A combinational function equal to the 9-bit sum of HPROD and MCND if the low-order bit of MPY/LPROD is 1, and equal to HPROD (extended to 9 bits) otherwise.

The MPY/LPROD shift register serves a dual purpose, holding both yet-to-be-tested multiplier bits (on the right) and unchanging product bits (on the left) as the algorithm is executed. At each step it shifts right one bit, discarding the multiplier bit that was just tested, moving the next multiplier bit to be tested to the rightmost position, and loading into the leftmost position one more product bit that will not change for the rest of the algorithm.

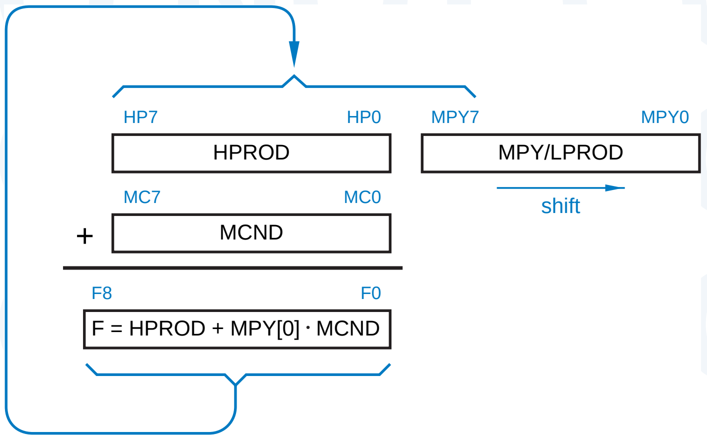


Figure 8-81 Registers and functions used by the shift-and-add multiplication algorithm.

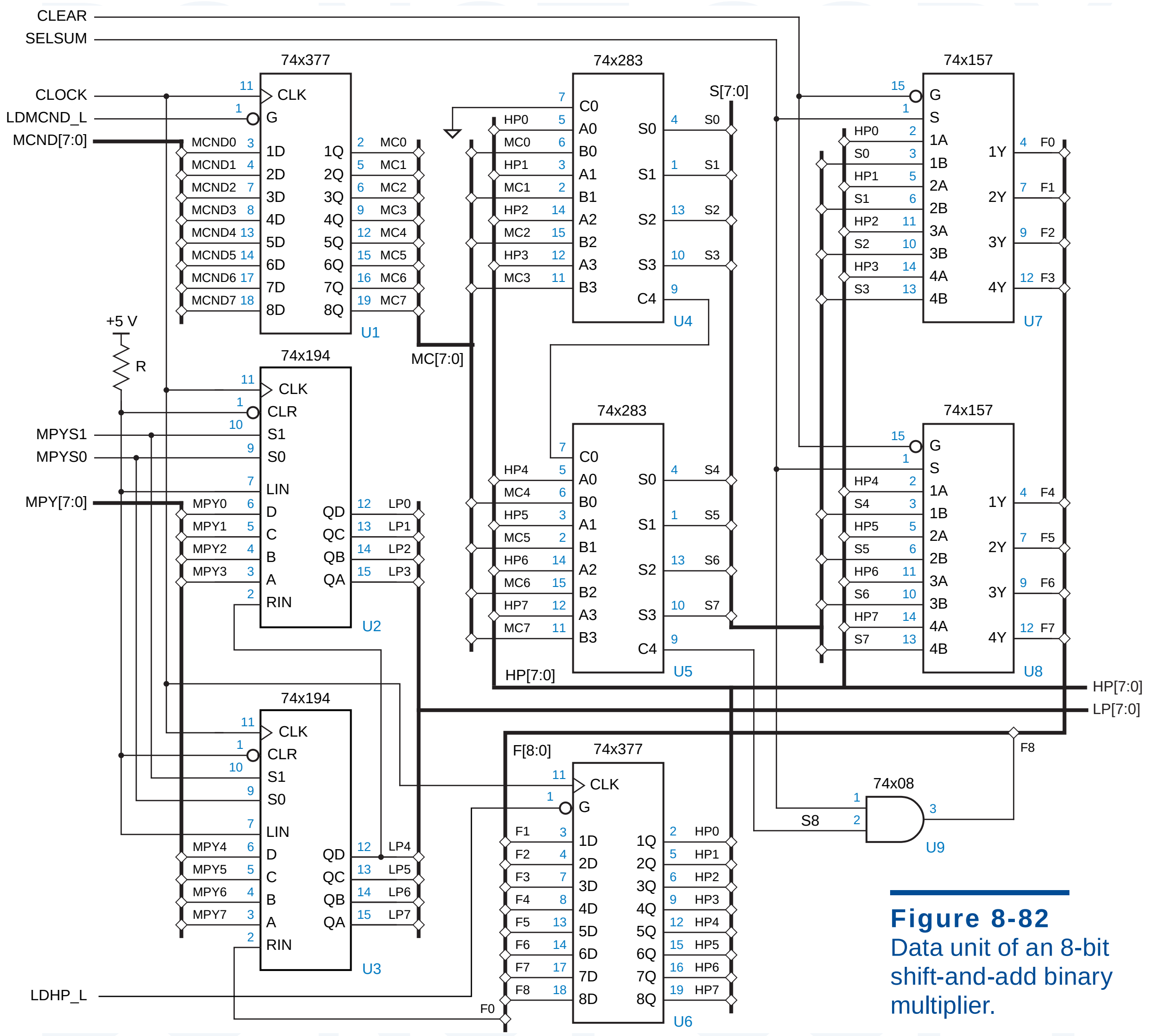


Figure 8-82 Data unit of an 8-bit shift-and-add binary multiplier.

Figure 8-82 is an MSI design for the data unit. The multiplier, MPY[7:0], and the multiplicand, MCND[7:0], are loaded into two registers before a multiplication begins. When the multiplication is completed, the product appears on HP[7:0] and LP[7:0]. The data unit uses the following control signals:

LDMCND_L	When asserted, enables the multiplicand register U1 to be loaded.
LDHP_L	When asserted, enables the HPROD register U6 to be loaded.
MPYS[1:0]	When 11, these signals enable the MPY/LPROD register U2 and U3 to be loaded at the next clock tick. They are set to 01 during the multiplication operation to enable the register to shift right, and are 00 at other times to preserve the register's contents.

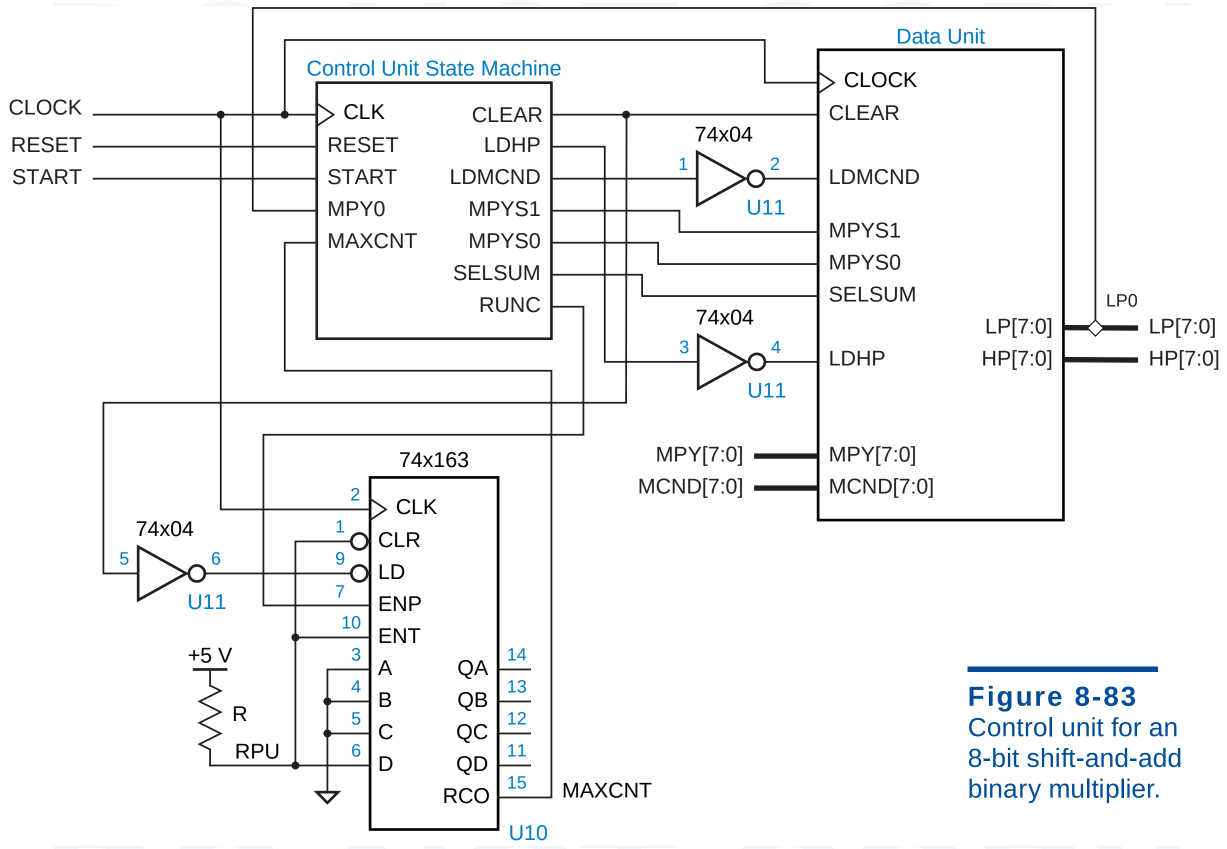


Figure 8-83
Control unit for an 8-bit shift-and-add binary multiplier.

SELSUM When this is asserted, the multiplexers U7 and U8 select the output of the adders U4 and U5, which is the sum of HPROD and the multiplicand MC. Otherwise, they select HPROD directly.

CLEAR When asserted, the output of multiplexers U7 and U8 is zero.

The multiplier uses a control unit, shown along with the data-unit block in Figure 8-83, to initialize the data unit and step through a multiplication. The control unit is decomposed into a counter (U10) and a state machine with the state diagram shown in Figure 8-84.

The state machine has the following inputs and outputs:

RESET A reset input that is asserted at power-up.

START An external command input that starts a multiplication.

MPY0 A condition input from the data unit, the next multiplier bit to test.

CLEAR A control output that zeroes the multiplexer output and initializes the counter.

LDMCND A control output that enables the MCND register to be loaded.

LDHP A control output that enables the HPROD register to be loaded.

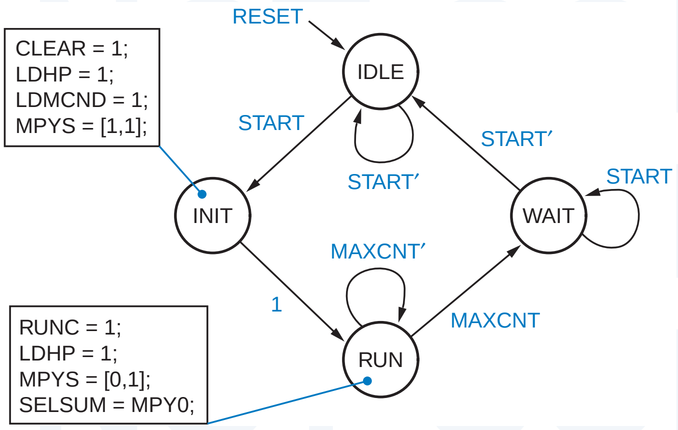


Figure 8-84 State diagram for the control state machine for a shift-and-add binary multiplier.

RUNC	A control output that enables the counter to count.
MPYS[1:0]	Control outputs for MPY/LPROD shifting and loading.
SELSUM	A control output that selects between the shifted adder output or shifted HPROD to be loaded back into HPROD.

The state diagram can be converted into a corresponding state machine using any of a variety of methods, from turn-the-crank (a.k.a. hand-crafted) design to automatic synthesis using a corresponding ABEL or VHDL description. The state machine has mostly Moore-type outputs; SELSUM is a Mealy-type output. Two boxes in the state diagram list outputs that are asserted in the INIT and RUN states; all outputs are negated at other times. The machine is designed so that asserting RESET in any state takes it to the IDLE state.

After the START signal is asserted, a multiplication begins in the INIT state. In this state, the counter is initialized to 1000_2, the multiplier and multiplicand are loaded into their respective registers, and HPROD is cleared. The RUN state is entered next, and the counter is enabled to count. The state machine stays in the RUN state for eight clock ticks, to execute the eight steps of the 8-bit shift-and-add algorithm. During the eighth tick, the counter is in state 1111_2, so MAXCNT is asserted and the state machine goes to the WAIT state. The machine waits there until START is negated, to prevent a multiplication from restarting until START is asserted once again.

The design details of the data and control units are interesting, but the most important thing to see in this example is that all of the sequential circuit elements for both data and control are edge-triggered flip-flops clocked by the same, common CLOCK signal. Thus, its timing is consistent with the model in Figure 8-80, and the designer need not be concerned about races, hazards, and asynchronous operations. Unless the state machine realization is very slow, the overall circuit's maximum clock speed will be limited mainly by the propagation delays through the data unit.

8.8 Impediments to Synchronous Design

Although the synchronous approach is the most straightforward and reliable method of digital system design, a few nasty realities can get in the way. We'll discuss them in this section.

8.8.1 Clock Skew

Synchronous systems using edge-triggered flip-flops work properly only if all flip-flops see the triggering clock edge at the same time. Figure 8-85 shows what can happen otherwise. Here, two flip-flops are theoretically clocked by the same signal, but the clock signal seen by FF2 is delayed by a significant amount relative to FF1's clock. This difference between arrival times of the clock at different devices is called *clock skew.*

clock skew

We've named the delayed clock in Figure 8-85(a) "CLOCKD." If FF1's propagation delay from CLOCK to Q1 is short, and if the physical connection of Q1 to FF2 is short, then the change in Q1 caused by a CLOCK edge may actually reach FF2 *before* the corresponding CLOCKD edge. In this case, FF2 may go to an incorrect next state determined by the *next* state of FF1 instead of the current state, as shown in (b). If the change in Q1 arrives at FF2 only slightly early relative to CLOCKD, then FF2's hold-time specification may be violated, in which case FF2 may become metastable and produce an unpredictable output.

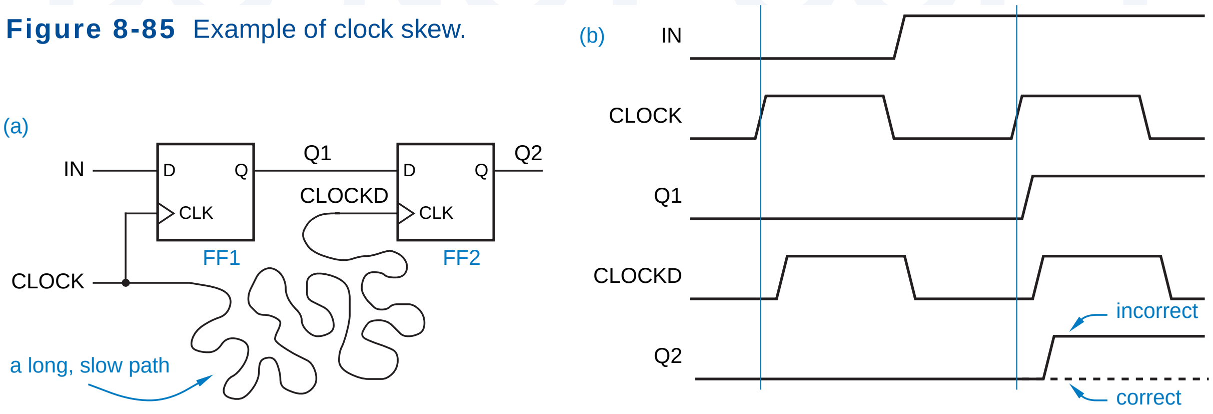
If Figure 8-85 reminds you of the essential hazard shown in Figure 7-101, you're on to something. The clock-skew problem may be viewed simply as a manifestation of the essential hazards that exist in all edge-triggered devices.

We can determine quantitatively whether clock skew is a problem in a given system by defining t_{skew} to be the amount of clock skew and using the other timing parameters defined in Figure 8-1. For proper operation, we need

$$t_{\mathrm{ffpd(min)}} + t_{\mathrm{comb(min)}} - t_{\mathrm{hold}} - t_{\mathrm{skew(max)}} > 0$$

In other words, clock skew subtracts from the hold-time margin that we defined in Section 8.1.4.

Figure 8-85 Example of clock skew.



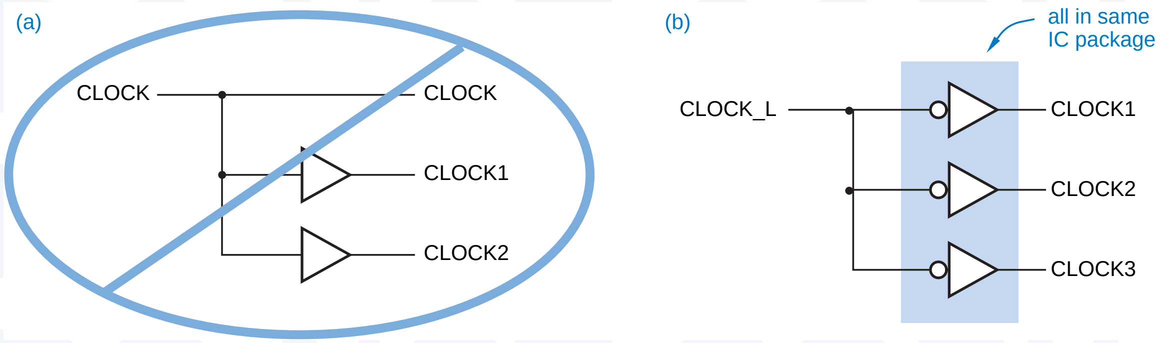


Figure 8-86 Buffering the clock: (a) excessive clock skew; (b) controllable clock skew.

Viewed in isolation, the example in Figure 8-85 may seem a bit extreme. After all, why would a designer provide a short connection path for data and a long one for the clock, when they could just run side by side? There are several ways this can happen; some are mistakes, while others are unavoidable.

In a large system, a single clock signal may not have adequate fanout to drive all of the devices with clock inputs, so it may be necessary to provide two or more copies of the clock signal. The buffering method of Figure 8-86(a) obviously produces excessive clock skew, since CLOCK1 and CLOCK2 are delayed through an extra buffer compared to CLOCK.

A recommended buffering method is shown in Figure 8-86(b). All of the clock signals go through identical buffers, and thus have roughly equal delays. Ideally, all the buffers should be part of the same IC package, so that they all have similar delay characteristics and are operating at identical temperature and power-supply voltage. Some manufacturers build special buffers for just this sort of application and specify the worst-case delay variation between buffers in the same package, which can be as low as a few tenths of a nanosecond.

Even the method in Figure 8-86(b) may produce excessive clock skew if one clock signal is loaded much more heavily than the other; transitions on the more heavily loaded clock appear to occur later because of increases in output-transistor switching delay and signal rise and fall times. Therefore, a careful designer tries to balance the loads on multiple clocks, looking at both DC load (fanout) and AC load (wiring and input capacitance).

Another bad situation can occur when signals on a PCB or in an ASIC are routed automatically by CAD software. Figure 8-87 shows a PCB or ASIC with many flip-flops and larger-scale elements, all clocked with a common CLOCK signal. The CAD software has laid out CLOCK in a serpentine path that winds its way past all the clocked devices. Other signals are routed point-to-point between an output and a small number of inputs, so their paths are shorter. To make matters worse, in an ASIC some types of “wire” may be slower than others (polysilicon vs. metal in CMOS technology). As a result, a CLOCK edge may indeed arrive at FF2 quite a bit later than the data change that it produces on Q1.

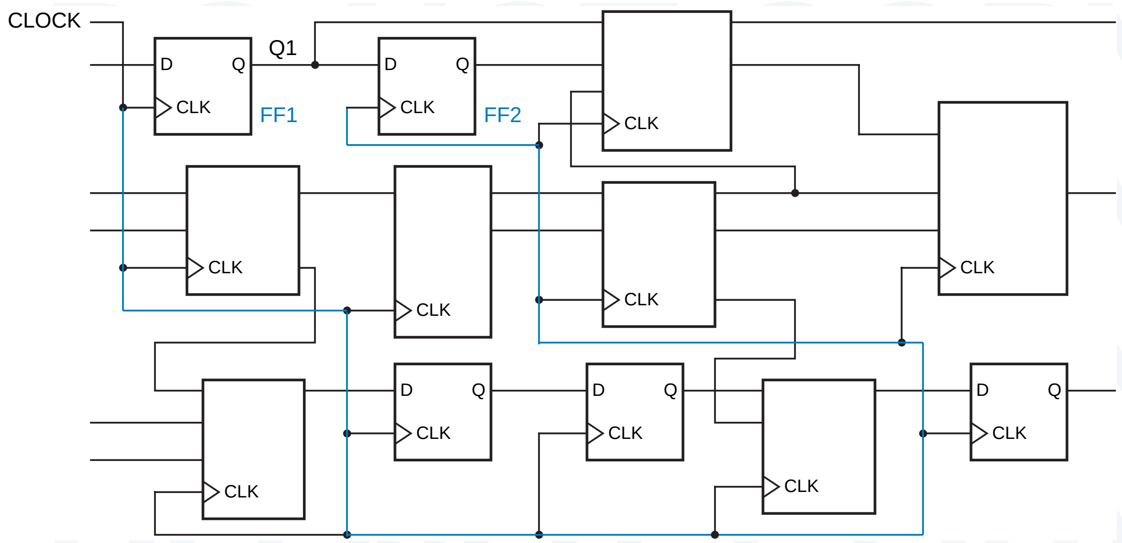
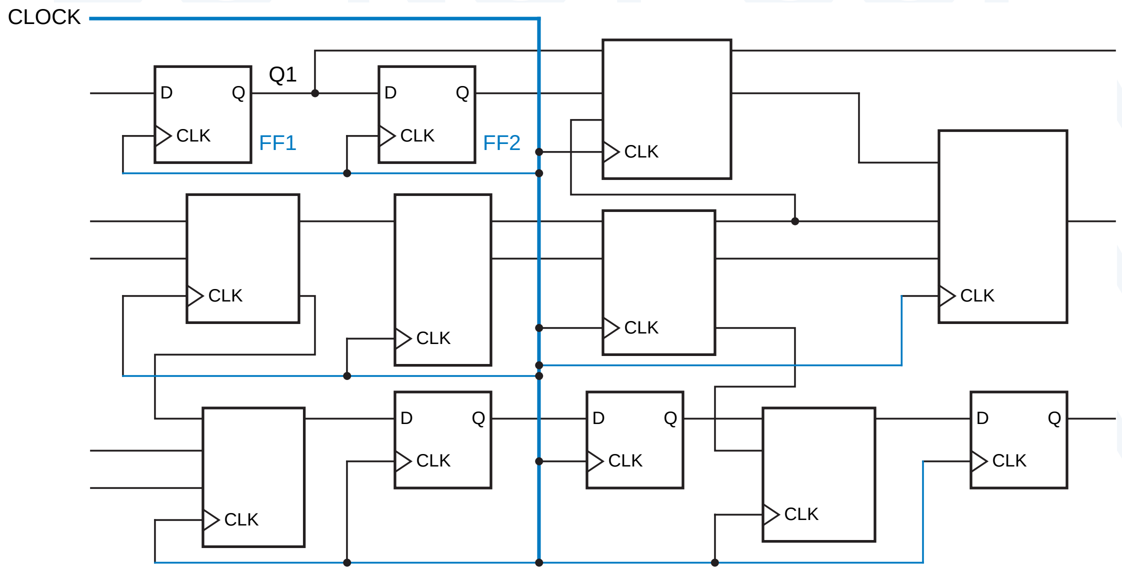


Figure 8-87 A clock-signal path leading to excessive skew in a complex PCB or ASIC.

One way to minimize this sort of problem is to arrange for CLOCK to be distributed in a tree-like structure using the fastest type of wire, as illustrated in Figure 8-88. Usually, such a "clock tree" must be laid out by hand or using a specialized CAD tool. Even then, in a complex design it may not be possible to guarantee that clock edges arrive everywhere before the earliest data change. A CAD timing analysis program is typically used to detect these problems, which

Figure 8-88 Clock-signal routing to minimize skew.



HOW NOT TO GET SKEWERED

Unbalanced wire lengths and loads are the most obvious sources of clock skew, but there are many other subtle sources. For example, *crosstalk*, the coupling of energy from one signal line into another, can cause clock skew. Crosstalk is inevitable when parallel wires are packed together tightly on a printed circuit board or in a chip, and energy is radiated during signal transitions. Depending on whether an adjacent signal is changing in the same or opposite direction as a clock, the clock's transition can be accelerated or retarded, making its transition appear to occur earlier or later.

In a large PCB or ASIC design, it's usually not feasible to track down all the possible sources of clock skew. As a result, most ASIC manufacturers require designers to provide extra setup- and hold-time margin, equivalent to many gate delays, over and above the known simulation timing results to accommodate such unknown factors.

generally can be remedied only by inserting extra delay (e.g., pairs of inverters) in the too-fast data paths.

two-phase latch design

Although synchronous design methodology simplifies the conceptual operation of large systems, we see that clock skew can be a major problem when edge-triggered flip-flops are used as the storage elements. To control this problem, many high-performance systems and VLSI chips use a *two-phase latch design*, discussed in the References. Such designs effectively split each edge-triggered D flip-flop into its two component latches, and control them with two nonoverlapping clock phases. The nonoverlap between the phases accommodates clock skew.

8.8.2 Gating the Clock

Most of the sequential MSI parts that we introduced in this chapter have synchronous function-enable inputs. That is, their enable inputs are sampled on the clock edge, along with the data. The first example that we showed was the 74x377 register with synchronous load-enable input; other parts included the 74x163 counter and 74x194 shift register with synchronous load-enable, count-enable, and shift-enable inputs. Nevertheless, many MSI parts, FPGA macros, and ASIC cells do not have synchronous function-enable inputs; for example, the 74x374 8-bit register has three-state outputs but no load-enable input.

gating the clock

So, what can a designer do if an application requires an 8-bit register with both a load-enable input *and* three-state outputs? One solution is to use a 74x377 to get the load-enable, and follow it with a 74x241 three-state buffer. However, this increases both cost and delay. Another approach is to use a larger, more expensive part, the 74x823, which provides both required functions as well as an asynchronous CLR_L input. A riskier but sometimes-used alternative is to use a '374, but to suppress its clock input when it's not supposed to be loaded. This is called *gating the clock*.

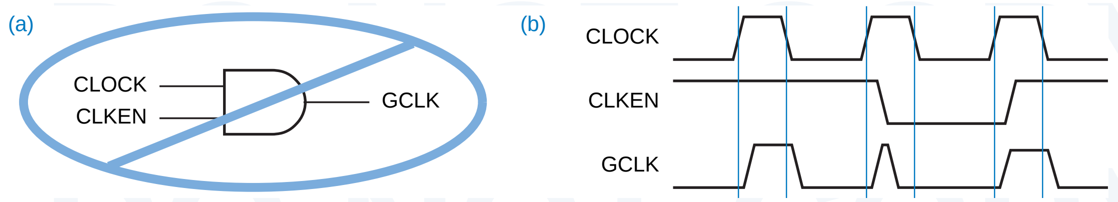


Figure 8-89 How not to gate the clock: (a) simple-minded circuit; (b) timing diagram.

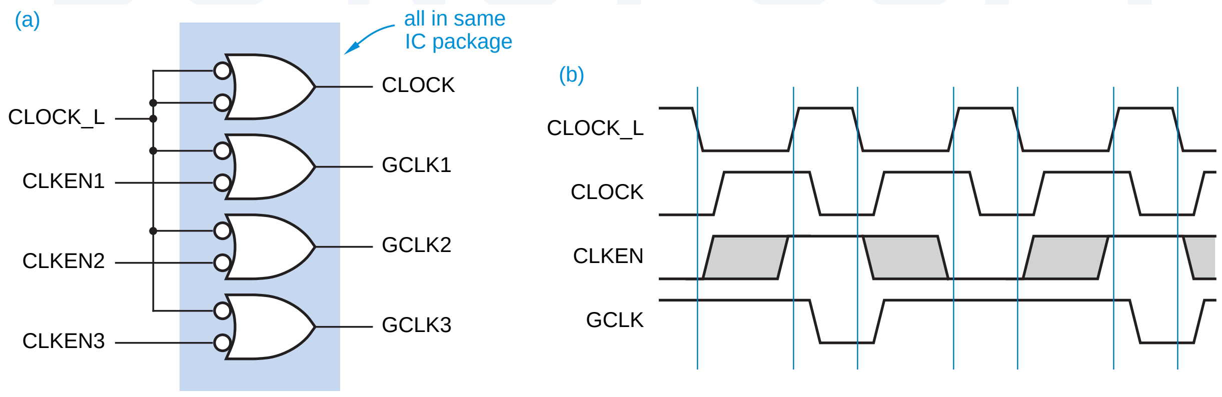
Figure 8-89 illustrates an obvious but wrong approach to gating the clock. A signal CLKEN is asserted to enable the clock, and is simply ANDed with the clock to produce the gated clock GCLK. This approach has two problems:

1. If CLKEN is a state-machine output or other signal produced by a register clocked by CLOCK, then CLKEN changes some time *after* CLOCK has already gone HIGH. As shown in (b) this produces glitches on GCLK, and false clocking of the registers controlled by GCLK.
2. Even if CLKEN is somehow produced well in advance of CLOCK's rising edge (e.g., using a register clocked with the *falling* edge of CLOCK, an especially nasty kludge), the AND-gate delay gives GCLK excessive clock skew, which causes more problems all around.

A method of gating the clock that generates only minimal clock skew is shown in Figure 8-90. Here, both an ungated clock and several gated clocks are generated from the same active-low master clock signal. Gates in the same IC package are used to minimize the possible differences in their delays. The CLKEN signal may change arbitrarily whenever CLOCK_L is LOW, which is when CLOCK is HIGH. That's just fine; a CLKEN signal is typically produced by a state machine whose outputs change right after CLOCK goes HIGH.

The approach of Figure 8-90 is acceptable in a particular application only if the clock skew that it creates is acceptable. Furthermore, note that CLKEN

Figure 8-90 An acceptable way to gate the clock: (a) circuit; (b) timing diagram.



must be stable during the entire time that CLOCK_L is HIGH (CLOCK is LOW). Thus, the timing margins in this approach are sensitive to the clock's duty cycle, especially if CLKEN suffers significant combinational-logic delay (t_{comb}) from the triggering clock edge. A truly synchronous function-enable input, such as the 74x377's load-enable input in Figure 8-13, can be changed at almost any time during the entire clock period, up until a setup time before the triggering edge.

8.8.3 Asynchronous Inputs

Even though it is theoretically possible to build a computer system that is fully synchronous, you couldn't do much with it, unless you can synchronize your keystrokes with a 500 MHz clock. Digital systems of all types inevitably must deal with *asynchronous input signals* that are not synchronized with the system clock.

asynchronous input signal

Asynchronous inputs are often requests for service (e.g., interrupts in a computer) or status flags (e.g., a resource has become available). Such inputs normally change slowly compared to the system clock frequency, and need not be recognized at a particular clock tick. If a transition is missed at one clock tick, it can always be detected at the next one. The transition rates of asynchronous signals may range from less than one per second (the keystrokes of a slow typist) to 100 MHz or more (access requests for a 500-MHz multiprocessor system's shared memory).

synchronizer

Ignoring the problem of metastability, it is easy to build a *synchronizer*, a circuit that samples an asynchronous input and produces an output that meets the setup and hold times required in a synchronous system. As shown in Figure 8-91, a D flip-flop samples the asynchronous input at each tick of the system clock and produces a synchronous output that is valid during the next clock period.

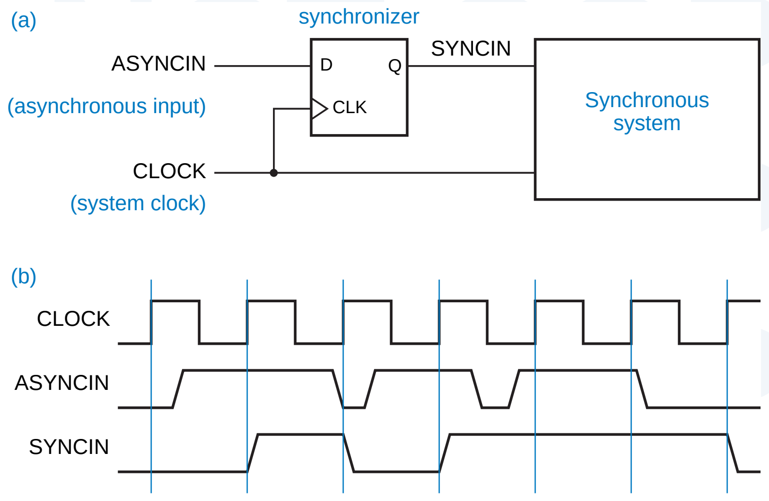


Figure 8-91 A single, simple synchronizer: (a) logic diagram; (b) timing.

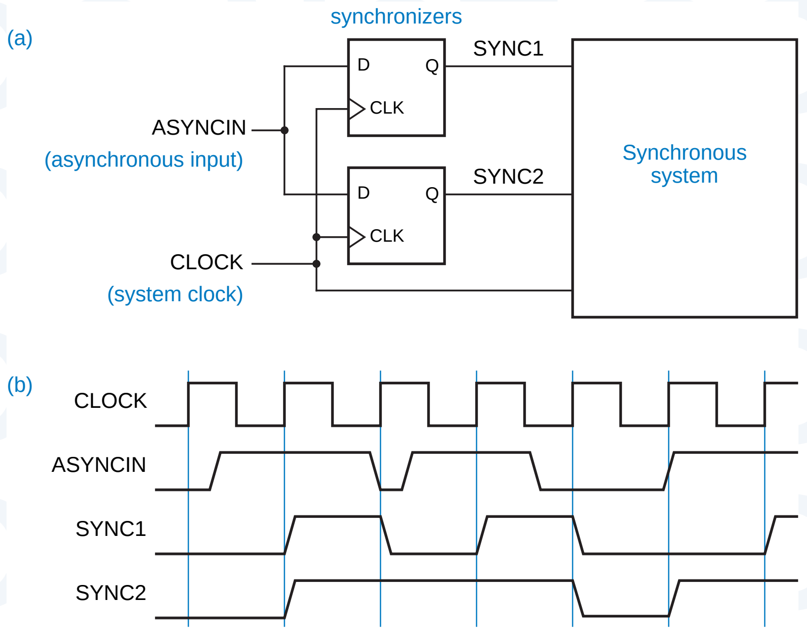
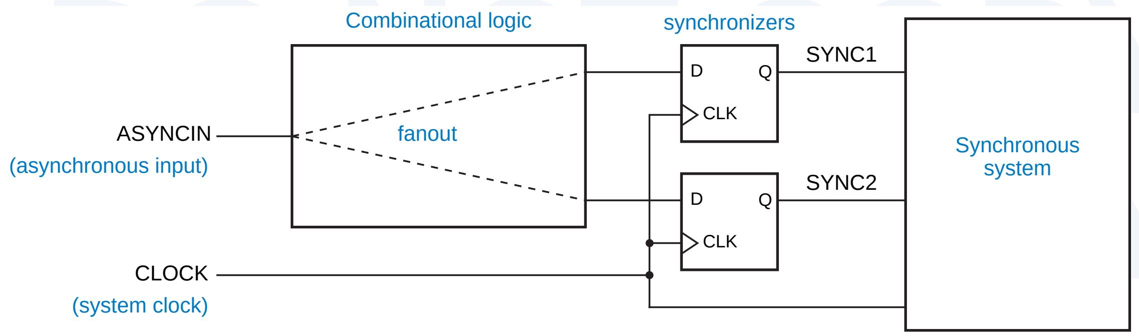


Figure 8-92 Two synchronizers for the same asynchronous input: (a) logic diagram; (b) possible timing.

It is essential for asynchronous inputs to be synchronized at only *one place* in a system; Figure 8-92 shows what can happen otherwise. Because of physical delays in the circuit, the two flip-flops will not see the clock and input signals at precisely the same time. Therefore, when asynchronous input transitions occur near the clock edge, there is a small window of time during which one flip-flop may sample the input as 1 and the other may sample it as 0. This inconsistent result may cause improper system operation, as one part of the system responds as if the input were 1, and another part responds as if it were 0.

Combinational logic may hide the fact that there are two synchronizers, as shown in Figure 8-93. Since different paths through the combinational logic will inevitably have different delays, the likelihood of an inconsistent result is even

Figure 8-93 An asynchronous input driving two synchronizers through combinational logic.



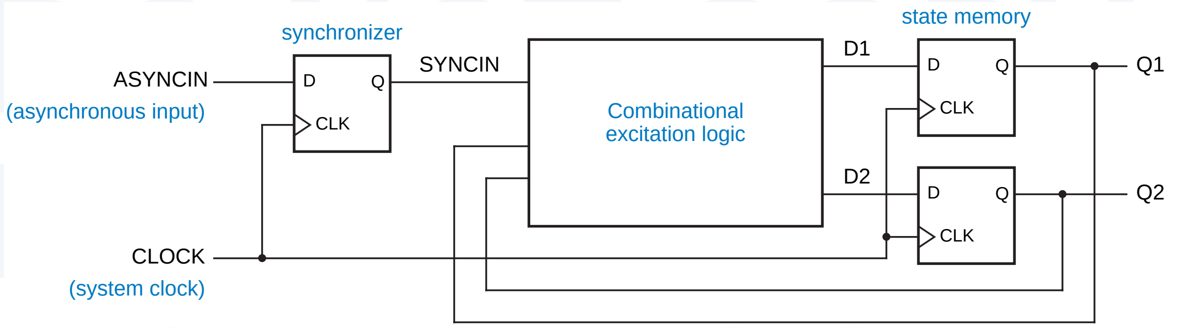


Figure 8-94 An asynchronous state-machine input coupled through a single synchronizer.

greater. This situation is especially common when asynchronous signals are used as inputs to state machines, since the excitation logic for two or more state variables may depend on the asynchronous input. The proper way to use an asynchronous signal as a state-machine input is shown in Figure 8-94. All of the excitation logic sees the same synchronized input signal, SYNCIN.

WHO CARES? As you probably know, even the synchronizers in Figures 8-91 and 8-94 sometimes fail. The reason they fail is that the setup and hold times of the synchronizing flip-flop are sometimes violated because the asynchronous input can change at any time. “Well, who cares?” you may say. “If the D input changes near the clock edge, then the flip-flop will either see the change this time or miss it and pick it up next time; either way is good enough for me!” The problem is, there is a third possibility, discussed in the next section.

8.9 Synchronizer Failure and Metastability

We showed in Section 7.1 that when the setup and hold times of a flip-flop are not met, the flip-flop may go into a third, *metastable* state halfway between 0 and 1. Worse, the length of time it may stay in this state before falling back into a legitimate 0 or 1 state is theoretically unbounded. When other gates and flip-flops are presented with a metastable input signal, some may interpret it as a 0 and others as a 1, creating the sort of inconsistent behavior that we showed in Figure 8-92. Or the other gates and flip-flops may produce metastable outputs themselves (after all, they are now operating in the *linear* part of their operating range). Luckily, the probability of a flip-flop output remaining in the metastable state decreases exponentially with time, though never all the way to zero.

8.9.1 Synchronizer Failure

Synchronizer failure is said to occur if a system uses a synchronizer output while the output is still in the metastable state. The way to avoid synchronizer failure is to ensure that the system waits “long enough” before using a synchronizer’s output, “long enough” so that the mean time between synchronizer failures is several orders of magnitude longer than the designer’s expected length of employment.

synchronizer failure

Metastability is more than an academic problem. More than a few experienced designers of high-speed digital systems have built (and released to production) circuits that suffer from intermittent synchronizer failures. In fact, the initial versions of several commercial ICs are said to have suffered from metastability problems, for example, the AMD 9513 system timing controller, the AMD 9519 interrupt controller, the Zilog Z-80 Serial I/O interface, the Intel 8048 single-chip microcomputer, and the AMD 29000 RISC microprocessor. It makes you wonder, are the designers of these parts still employed?

There are two ways to get a flip-flop out of the metastable state:

1. Force the flip-flop into a valid logic state using input signals that meet the published specifications for minimum pulse width, setup time, and so on.
2. Wait “long enough,” so the flip-flop comes out of metastability on its own.

Inexperienced designers often attempt to get around metastability in other ways, and they are usually unsuccessful. Figure 8-95 shows an attempt by a designer who thinks that since metastability is an “analog” problem, it must have an “analog” solution. After all, Schmitt trigger inputs and capacitors can normally be used to clean up noisy signals. However, rather than eliminate metastability, this circuit enhances it—with the “right” components, the circuit will oscillate forever once it is excited by negating S_L and R_L simultaneously. (*Confession:* It was the author who tried this over 20 years ago!) Exercises 8.74 and 8.75 give examples of valiant but also failed attempts to eliminate metastability. These examples should give you the sense that synchronizer problems can be very subtle, so you must be careful. The only way to make synchronizers reliable is to wait long enough for metastable outputs to resolve. We answer the question “How long is ‘long enough’?” later in this section.

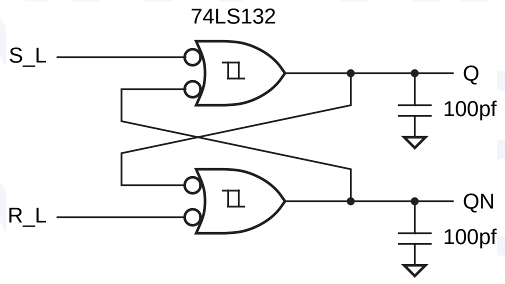


Figure 8-95
A failed attempt to build a metastable-proof $\overline{S}$-$\overline{R}$ flip-flop.

8.9.2 Metastability Resolution Time

If the setup and hold times of a D flip-flop are met, the flip-flop output settles to a new value within time t_{pd} after the clock edge. If they are violated, the flip-flop output may be metastable for an arbitrary length of time. In a particular system design, we use the parameter t_r, called the *metastability resolution time*, to denote the maximum time that the output can remain metastable without causing synchronizer (and system) failure.

t_r
metastability resolution time

For example, consider the state machine in Figure 8-94 on page 666. The available metastability resolution time is

$$t_r = t_{clk} - t_{comb} - t_{setup}$$

where t_{clk} is the clock period, t_{comb} is the propagation delay of the combinational excitation logic, and t_{setup} is the setup time of the flip-flops used in the state memory.

t_{clk}
t_{comb}
t_{setup}

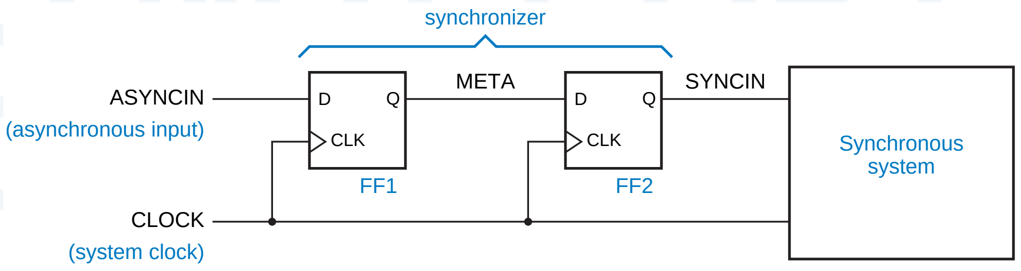
8.9.3 Reliable Synchronizer Design

The most reliable synchronizer is one that allows the maximum amount of time for metastability resolution. However, in the design of a digital system, we seldom have the luxury of *slowing down* the clock to make the system work more reliably. Instead, we are usually asked to *speed up* the clock to get higher performance from the system. As a result, we often need synchronizers that work reliably with very short clock periods. We'll show several such designs, and show how to predict their reliability.

We showed previously that a state machine with an asynchronous input, built as shown in Figure 8-94, has $t_r = t_{clk} - t_{comb} - t_{setup}$. In order to maximize t_r for a given clock period, we should minimize t_{comb} and t_{setup}. The value of t_{setup} depends on the type of flip-flops used in the state memory; in general, faster flip-flops have shorter setup times. The minimum value of t_{comb} is zero, and is achieved by the synchronizer design of Figure 8-96, whose operation we explain next.

Inputs to flip-flop FF1 are asynchronous with the clock, and may violate the flip-flop's setup and hold times. When this happens, the META output may become metastable and remain in that state for an arbitrary time. However, we

Figure 8-96 Recommended synchronizer design.



assume that the maximum duration of metastability after the clock edge is t_r. (We show how to calculate the probability that our assumption is correct in the next subsection.) As long as the clock period is greater than t_r plus the FF2's setup time, SYNCIN becomes a synchronized copy of the asynchronous input on the next clock tick without ever becoming metastable itself. The SYNCIN signal is distributed as required to the rest of the system.

8.9.4 Analysis of Metastable Timing

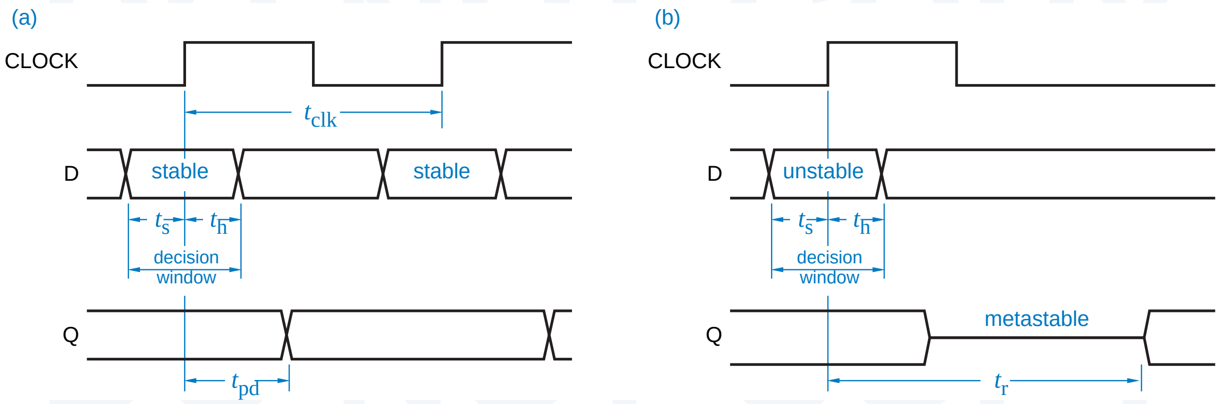
Figure 8-97 shows the flip-flop timing parameters that are relevant to our analysis of metastability timing. The published setup and hold times of a flip-flop with respect to its clock edge are denoted by t_s and t_h, and bracket an interval called the *decision window*, when the flip-flop samples its input and decides to change its output if necessary. As long as the D input changes outside the decision window, as in (a), the manufacturer guarantees that the output will change and settle to a valid logic state before time t_{pd}. If D changes inside the decision window, as in (b), metastability may occur and persist until time t_r.

decision window

Theoretical research suggests and experimental research has confirmed that when asynchronous inputs change during the decision window, the duration of metastable outputs is governed by an exponential formula:

$$\mathrm{MTBF}(t_r) = \frac{\exp(t_r/\tau)}{T_o \cdot f \cdot a}$$

Figure 8-97 Timing parameters for metastability analysis: (a) normal flip-flop operation; (b) metastable behavior.



DETAILS, DETAILS

In our analysis of the synchronizer in Figure 8-96, we do not allow metastability, even briefly, on the output of FF2, because we assume that the system has been designed with zero timing margins. If the system can in fact tolerate some increase in FF2's propagation delay, the MTBF will be somewhat better than predicted.

f
a
T_o
τ

Here MTBF(t_r) is the mean time between synchronizer failures, where a failure occurs if metastability persists beyond time t_r after a clock edge, where $t_r \geq t_{pd}$. This MTBF depends on f, the frequency of the flip-flop clock; a, the number of asynchronous input changes per second applied to the flip-flop; and T_o and τ, constants that depend on the electrical characteristics of the flip-flop. For a typical 74LS74, $T_o \approx 0.4$ s and $\tau \approx 1.5$ ns.

Now suppose that we build a microprocessor system with a 10 MHz clock and use the circuit of Figure 8-96 to synchronize an asynchronous input. If ASYNCIN changes during the decision window of FF1, the output META may become metastable until time t_r. If META is still metastable at the beginning of the decision window for FF2, then the synchronizer fails because FF2 may have a metastable output; system operation is unpredictable in that case.

Let us assume that the D flip-flops in Figure 8-96 are 74LS74s. The setup time t_s of a 74LS74 is 20 ns, and the clock period in our example microprocessor system is 100 ns, so t_r for synchronizer failure is 80 ns. If the asynchronous input changes 100,000 times per second, then the synchronizer MTBF is

$$\text{MTBF}(80\text{ ns}) = \frac{\exp(80/1.5)}{0.4 \cdot 10^7 \cdot 10^5} = 3.6 \cdot 10^{11}\text{ s}$$

That's not bad, about 100 centuries between failures! Of course, if we're lucky enough to sell 10,000 copies of our system, one of them will fail in this way every year. But, no matter, let us consider a more serious problem.

Suppose we upgrade our system to use a faster microprocessor chip with a clock speed of 16 MHz. We may have to replace some components in our system to operate at the higher speed, but 74LS74s are still perfectly good at 16 MHz. Or are they? With a clock period of 62.5 ns, the new synchronizer MTBF is

$$\text{MTBF}(42.5\text{ ns}) = \frac{\exp(42.5/1.5)}{0.4 \cdot 1.6 \cdot 10^7 \cdot 10^5} = 3.1\text{ s!}$$

UNDERSTANDING *A* AND *F*

Although a flip-flop output can go metastable *only* if D changes during the decision window, the MTBF formula does not explicitly specify how many such input changes occur. Instead, it specifies the total number of asynchronous input changes per second, a, and assumes that asynchronous input changes are uniformly distributed over the clock period. Therefore, the fraction of input changes that actually occur during the decision window is "built in" to the clock-frequency parameter f—as f increases, the fraction goes up.

If the system design is such that input changes might be clustered in the decision window rather than being uniformly distributed (as when synchronizing a slow input with a fixed but unknown phase difference from the system clock), then a useful rule of thumb is to use a frequency equal to the reciprocal of the decision window (based on published setup and hold times), times a safety margin of 10.

The only saving grace of this synchronizer at 16 MHz is that it's so lousy, we're likely to discover the problem in the engineering lab before the product ships! Thank goodness the MTBF wasn't one year.

8.9.5 Better Synchronizers

Given the poor performance of the 74LS74 as a synchronizer at moderate clock speeds, we have a couple of alternatives for building more reliable synchronizers. The simplest solution, which works for most design requirements, is simply to use a flip-flop from a faster technology. Nowadays there are available much faster technologies for flip-flops, whether discrete or embedded in PLDs, FPGAs, or ASICs.

Based on published data discussed in the References, Table 8-35 lists the metastability parameters for several common logic families and devices. These numbers are very much circuit-design and IC-process dependent. Thus, unlike guaranteed logic signal levels and timing parameters, published metastability numbers can vary dramatically among different manufacturers of the same part and must be used conservatively. For example, one manufacturer's 74F74 may give acceptable metastability performance in a design while another's may not.

Table 8-35 Metastability parameters for some common devices.

Reference	*Device*	τ *(ns)*	T_o *(s)*	t_r *(ns)*
Chaney (1983)	74LS74	1.50	$4.0 \cdot 10^{-1}$	77.71
Chaney (1983)	74S74	1.70	$1.0 \cdot 10^{-6}$	66.14
Chaney (1983)	74S174	1.20	$5.0 \cdot 10^{-6}$	48.62
Chaney (1983)	74S374	0.91	$4.0 \cdot 10^{-4}$	40.86
Chaney (1983)	74F74	0.40	$2.0 \cdot 10^{-4}$	17.68
TI (1997)	74LSxx	1.35	$4.8 \cdot 10^{-3}$	63.97
TI (1997)	74Sxx	2.80	$1.3 \cdot 10^{-9}$	90.33
TI (1997)	74ALSxx	1.00	$8.7 \cdot 10^{-6}$	41.07
TI (1997)	74ASxx	0.25	$1.4 \cdot 10^{3}$	14.99
TI (1997)	74Fxx	0.11	$1.9 \cdot 10^{8}$	7.90
TI (1997)	74HCxx	1.82	$1.5 \cdot 10^{-6}$	71.55
Cypress (1997)	PALC16R8-25	0.52	$9.5 \cdot 10^{-12}$	14.22*
Cypress (1997)	PALC22V10B-20	0.26	$5.6 \cdot 10^{-11}$	7.57*
Cypress (1997)	PALCE22V10-7	0.19	$1.3 \cdot 10^{-13}$	4.38*
Xilinx (1997)	7300-series CPLD	0.29	$1.0 \cdot 10^{-15}$	5.27*
Xilinx (1997)	9500-series CPLD	0.17	$9.6 \cdot 10^{-18}$	2.30*

Note that different authors and manufacturers may specify metastability parameters differently. For example, author Chaney and manufacturer Texas Instruments measure the metastability resolution time t_r from the triggering clock edge, as in our previous subsection. On the other hand, manufacturers Cypress and Xilinx define t_r as the *additional* delay beyond a normal clock-to-output delay time t_{pd}.

The last column in the table gives a somewhat arbitrarily chosen figure of merit for each device. It is the metastability resolution time t_r required to obtain an MTBF of 1000 years when operating a synchronizer with a clock frequency of 25 MHz and with 100,000 asynchronous input changes per second. For the Cypress and Xilinx devices, their parameter values yield a value of t_r, marked with an asterisk, consistent with their own definition as introduced above.

As you can see, the 74LS74 is one of the worst devices in the table. If we replace FF1 in the 16 MHz microprocessor system of the preceding subsection with a 74ALS74, we get

$$\text{MTBF(42.5 ns)} = \frac{\exp(42.5/0.87)}{12.5 \cdot 10^{-3} \cdot 1.6 \cdot 10^{7} \cdot 10^{5}} = 8.2 \cdot 10^{10}\,\text{s}$$

If you're satisfied with a synchronizer MTBF of about 25 centuries per system shipped, you can stop here. However, if FF2 is also replaced with a 74ALS74, the MTBF gets better, since the 'ALS74 has a shorter setup time than the 'LS74, only 10 ns. With the 'ALS74, the MTBF is about 100,000 times better:

$$\text{MTBF(52.5 ns)} = \frac{\exp(52.5/0.87)}{12.5 \cdot 10^{-3} \cdot 2 \cdot 10^{7} \cdot 10^{5}} = 8.1 \cdot 10^{15}\,\text{s}$$

Even if we ship a million systems containing this circuit, we (or our heirs) will see a synchronizer failure only once in 240 years. Now that's job security!

Actually, the margins above aren't as large as they might seem. (How large does 240 years seem *to you*?) Most of the numbers given in Table 8-35 are *averages*, and are seldom specified, let alone guaranteed, by the device manufacturer. Furthermore, calculated MTBFs are extremely sensitive to the value of τ, which in turn may depend on temperature, voltage, and the phase of the moon. So the operation of a given flip-flop in an actual system may be much worse (or much better) than predicted by our table.

For example, consider what happens if we increase the clock in our 16 MHz system by just 25%, to 20 MHz. Your natural inclination might be to think that metastability will get 25% worse, or maybe 250% worse, just to be conservative. But, if you run the numbers, you'll find that the MTBF using 'ALS74s for both FF1 and FF2 goes down from $8.1 \cdot 10^{15}$ s to just $3.7 \cdot 10^{9}$ s, over a million times worse! The new MTBF of about 118 years is fine for one system, but if you ship a million of them, one will fail every hour. You've just gone from generations of job security to corporate goat!

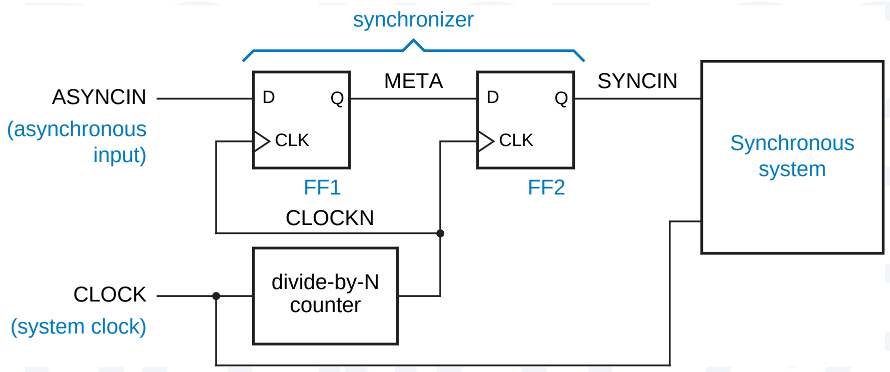


Figure 8-98 Multiple-cycle synchronizer.

8.9.6 Other Synchronizer Designs

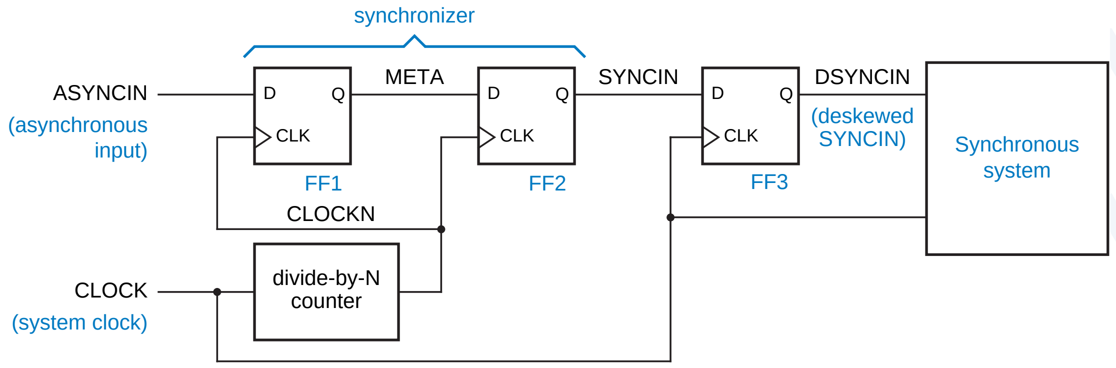
We promised to describe a couple of ways to build more reliable synchronizers. The first way was to use faster flip-flops, that is, to reduce the value of τ in the MTBF equation. Having said that, the second way is obvious—to *increase* the value of t_r in the MTBF equation.

multiple-cycle synchronizer

For a given system clock, the best value we can obtain for t_r using the circuit of Figure 8-96 is t_{clk}, if FF2 has a setup time of 0. However, we can get values of t_r on the order of $n \cdot t_{clk}$ by using the *multiple-cycle synchronizer* circuit of Figure 8-98. Here we divide the system clock by n to obtain a slower synchronizer clock and longer $t_r = (n \cdot t_{clk}) - t_{setup}$. Usually a value of $n = 2$ or $n = 3$ gives adequate synchronizer reliability.

In the figure, note that the edges of CLOCKN will lag the edges of CLOCK because CLOCKN comes from the Q output of a counter flip-flop that is clocked by CLOCK. This means that SYNCIN, in turn, will be delayed or skewed relative to other signals in the synchronous system that come directly from flip-flops clocked by CLOCK. If SYNCIN goes through additional combinational logic in the synchronous system before reaching its flip-flop inputs, their setup time may be inadequate. If this is the case, the solution in Figure 8-99 can be used. Here, SYNCIN is reclocked by CLOCK using FF3 to produce DSYNCIN, which will

Figure 8-99 Multiple-cycle synchronizer with deskewing.



have the same timing as other flip-flop outputs in the synchronous system. Of course, the delay from CLOCK to CLOCKN must still be short enough that SYNCIN meets the setup time requirement of FF3.

In an n-cycle synchronizer, the larger the value of n, the longer it takes for an asynchronous input change to be seen by the synchronous system. This is simply a price that must be paid for reliable system operation. In typical microprocessor systems, most asynchronous inputs are for external events—interrupts, DMA requests, and so on—that need not be recognized very quickly, relative to synchronizer delays. In the time-critical area of main memory access, experienced designers use the processor clock to run the memory subsystem too, if possible. This eliminates the need for synchronizers and provides the fastest possible system operation.

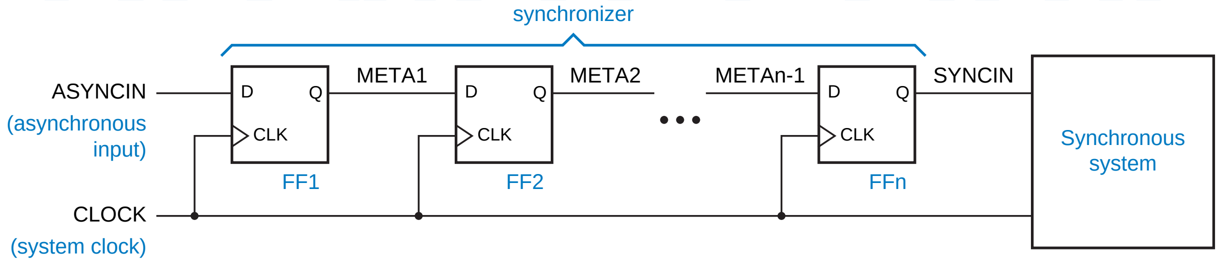
cascaded synchronizers

At higher frequencies, the feasibility of the multiple-cycle synchronizer design shown in Figure 8-98 tends to be limited by clock skew. For this reason, rather than use a divide-by-n synchronizer clock, some designers use *cascaded synchronizers.* This design approach simply uses a cascade (shift register) of n flip-flops, all clocked with the high-speed system clock. This approach is shown in Figure 8-100.

With cascaded synchronizers, the idea is that metastability will be resolved with some probability by the first flip-flop, and failing that, with an equal probability by each successive flip-flop in the cascade. So the overall probability of failure is on the order of the nth power of the failure probability of a single-flip-flop synchronizer at the system clock frequency. While this is partially true, the MTBF of the cascade is poorer than that of a multiple-cycle synchronizer with the same delay ($n \cdot t_{\text{clk}}$). With the cascade, the flip-flop setup time t_{setup} must be subtracted from t_r, the available metastability resolution time, n times, but in a multiple-cycle design, it is subtracted only once.

PLDs that contain internal flip-flops can be used in synchronizer designs, where the two flip-flops in Figure 8-96 on page 668 are simply included in the PLD. This is very convenient in most applications, because it eliminates the need for external, discrete flip-flops. However, a PLD-based synchronizer typically has a poorer MTBF than a discrete circuit built with the same or similar

Figure 8-100 Cascaded synchronizer.



technology. This happens because each flip-flop in a PLD has a combinational logic array on its D input that increases its setup time and thereby reduce the amount of time t_r available for metastability resolution during a given system clock period t_{clk}. To maximize t_r without using special components, FF2 in Figure 8-96 should be a short-setup-time discrete flip-flop.

8.9.7 Metastable-Hardened Flip-Flops

In the late 1980s, Texas Instruments and other manufacturers developed SSI and MSI flip-flops that are specifically designed for board-level synchronizer applications. For example, the *74AS4374* was similar to the 74AS374, except that each individual flip-flop was replaced with a pair of flip-flops, as shown in Figure 8-101. Each pair of flip-flops could be used as a synchronizer of the type shown in Figure 8-96, so eight asynchronous inputs could be synchronized with one 74AS4374.

74AS4374

The internal design of the ’AS4374 was improved to reduce τ and T_o compared to other 74AS flip-flops, but the biggest improvement in the ’AS4374 was a greatly reduced t_{setup}. Because the entire synchronizer of Figure 8-96 is built on a single chip, there are no input or output buffers between FF1 and FF2, and t_{setup} for FF2 is only 0.5 ns. Compared to a conventional 74AS flip-flop with a 5 ns t_{setup}, and assuming that $\tau = 0.40$ ns, this improves the MTBF by a factor of exp(4.5/.40), or about 77,000.

In recent years, the move towards faster CMOS technologies and higher integration has largely obsoleted specialized parts like the ’AS4374. As you can see from the last few rows in Table 8-35 on page 671, fast PLDs and CPLDs are available with values of τ that rival the fastest discrete devices while offering the convenience of integrating synchronization with many other functions. Still, the approach used by ’AS4374 is worth emulating in FPGA and ASIC designs. That is, whenever you have control over the layout of a synchronizer circuit, it pays to locate FF1 and FF2 as close as possible to each other, and to connect them with the fastest available wires, in order to maximize the setup time available for FF2.

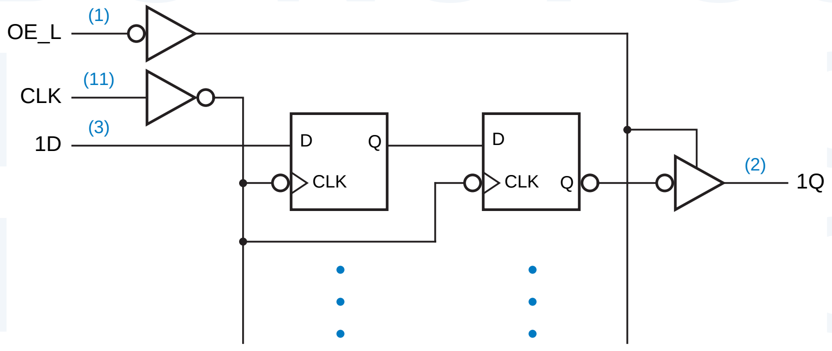


Figure 8-101 Logic diagram for the 74AS4374 octal dual-rank D flip-flop.

8.9.8 Synchronizing High-Speed Data Transfers

A very common problem in computer systems is synchronizing external data transfers with the computer system clock. A simple example is the interface between personal computer's network interface card and a 100 Mbps Ethernet link. The interface card may be connected to a PCI bus, which has a 33.33 MHz clock. Even though the Ethernet speed is an approximate multiple of the bus speed, the signal received on the Ethernet link is generated by another computer whose transmit clock is not synchronized with the receive clock in any way. Yet the interface must still deliver data reliably to the PCI bus.

Figure 8-102 shows the problem. NRZ serial data RDATA is received from the Ethernet at 100 Mbps. The digital phase-locked loop (DPLL) recovers a 100-MHz clock signal RCLK which is centered on the 100 Mbps data stream and

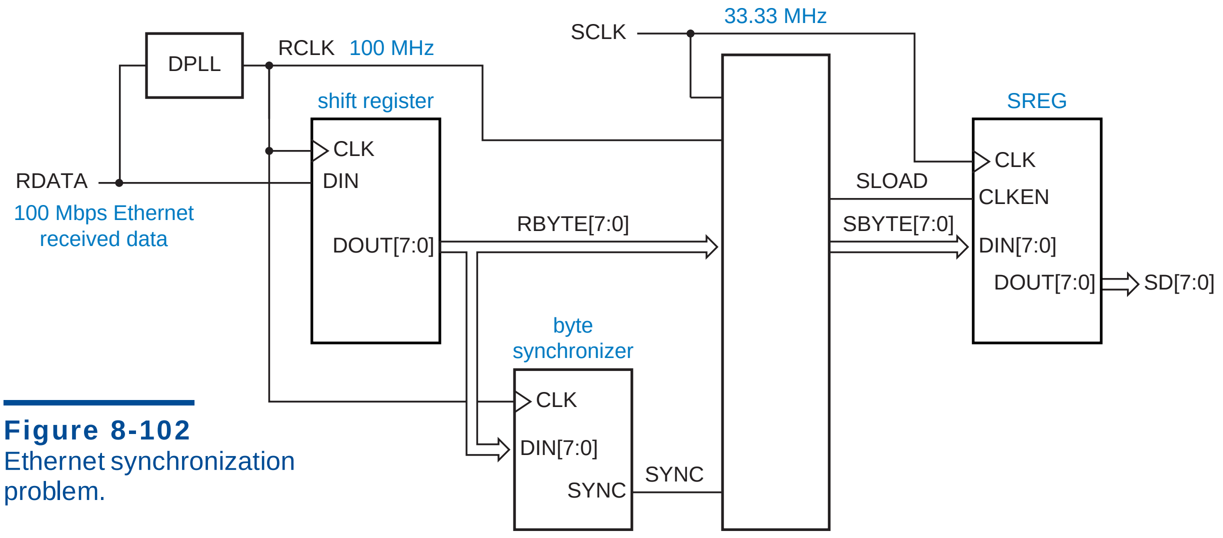


Figure 8-102
Ethernet synchronization problem.

ONE NIBBLE AT A TIME

The explanation of 100 Mbps Ethernet reception above is oversimplified, but sufficient for discussing the synchronization problem. In reality, the received data rate is 125 Mbps, where each 4 bits of user data is encoded as a 5-bit symbol using a so-called 4B5B code. By using only 16 out of 32 possible 5-bit codewords, the 4B5B code guarantees that regardless of the user data pattern, the bit stream on the wire will have a sufficient number of transitions to allow clock recovery. Also, the 4B5B code includes a special code that is transmitted periodically to allow nibble (4-bit) and byte synchronization to be accomplished very easily.

As a result of nibble synchronization, a typical 100-Mbps Ethernet interface does not see an unsynchronized 100 MHz stream of bits. Instead, it sees an unsynchronized 25 MHz stream of nibbles. So, the details of a real 100-Mbps Ethernet synchronizer are different, but the same principles apply.

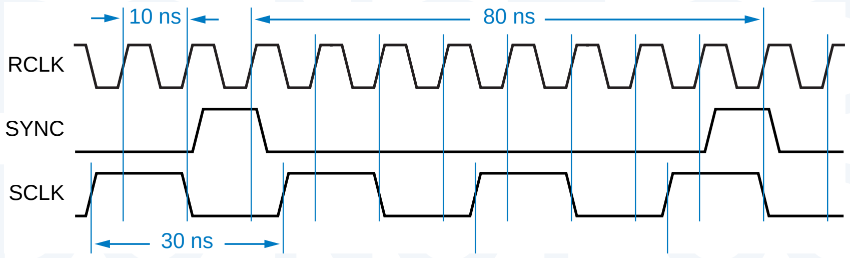


Figure 8-103 Ethernet link and system clock timing.

allows data to be clocked bit-by-bit into an 8-bit shift register. At the same time, a byte synchronization circuit searches for special patterns in the received data stream that indicate byte boundaries. When it detects one of these, it asserts the SYNC signal and does so on every eighth subsequent RCLK tick, so that SYNC is asserted whenever the shift register contains an aligned 8-bit byte. The rest of the system is clocked by a 33.33 MHz clock SCLK. We need to transfer each aligned byte RBYTE[7:0] into a register SREG in SCLK's domain. How can we do it?

Figure 8-103 shows some of the timing. We immediately see is that the byte-aligned signal, SYNC, is asserted for only 10 ns per byte. We have no hope of consistently detecting this signal with the asynchronous SCLK, whose period is a much longer 30 ns.

The strategy that is almost universally followed in this kind of situation is to transfer the aligned data first into a holding register HREG in the *receive* clock (RCLK) domain. This gives us a lot more time to sort things out, 80 ns in this case. Thus, the "?" box in Figure 8-102 can be replaced by Figure 8-104, which shows HREG and a box marked "SCTRL." The job of SCTRL is to assert SLOAD during exactly one 30-ns SCLK period, so that the output of HREG is valid and stable for the setup and hold times of register SREG in the SCLK domain. SLOAD also serves as a "new-data available" signal for the rest of the interface, indicating that a new data byte will appear on SBYTE[7:0] during the

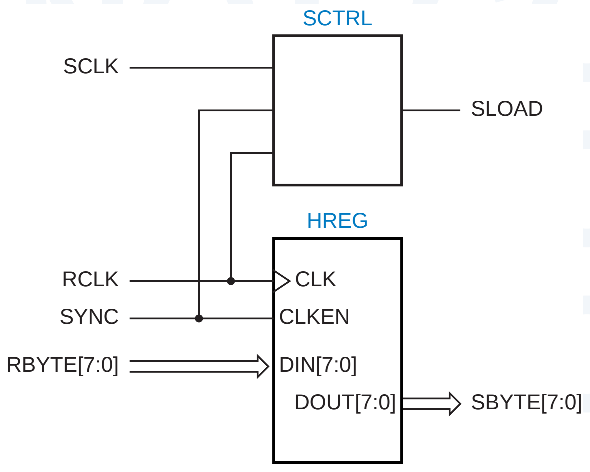


Figure 8-104 Byte holding register and control.

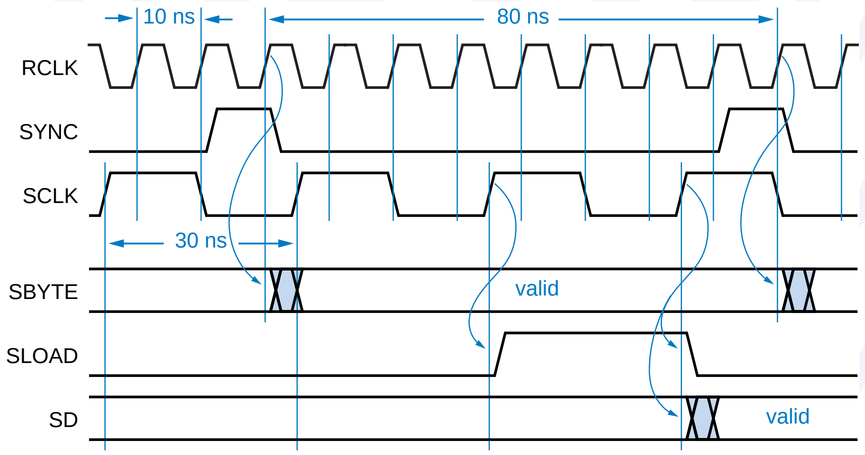


Figure 8-105 Timing for SBYTE and possible timing for SLOAD.

next SCLK period. Figure 8-105 shows possible timing for SLOAD based on this approach and the previous timing diagram.

Figure 8-106 is a circuit that can generate SLOAD with the desired timing. The idea is to use SYNC to set an S-R latch as a new byte becomes available. The output of this latch, NEWBYTE, is sampled by FF1 in the SCLK domain. Since NEWBYTE is not synchronized with SCLK, FF1's output SM may be metastable, but it is not used by FF2 until the next clock tick, 30 ns later. Assuming that the AND gate is reasonably fast, this gives plenty of metastability resolution time. FF2's output is the SLOAD signal. The AND gate ensures that SLOAD is only one SCLK period wide; if SLOAD is already 1, it can't be set to 1 on the next tick. This gives time for the S-R latch to be reset by SLOAD in preparation for the next byte.

A timing diagram for the overall circuit with "typical" timing is shown in Figure 8-107. Since SCLK is asynchronous to RCLK, it can have an arbitrary relationship with RCLK and SYNC. In the figure, we've shown a case where the next SCLK rising edge occurs well after NEWBYTE is set. Although the figure shows a window in which SM and SM1 could be metastable in the general case, metastability doesn't actually happen when the timing is as drawn. Later, we'll show what can happen if the SCLK edge occurs when NEWBYTE is changing.

We should make several notes about the circuit in Figure 8-106. First, the SYNC signal must be glitch-free, since it controls the S input of a latch, and it must be wide enough to meet the minimum pulse width requirement of the latch.

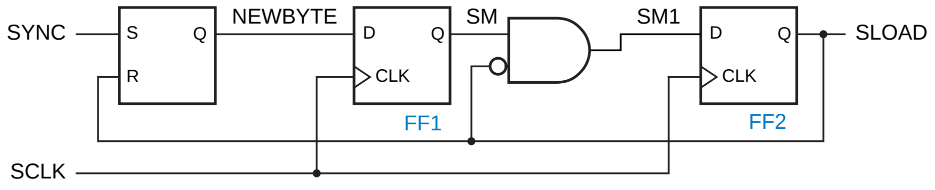


Figure 8-106 SCTRL circuit for generating SLOAD.

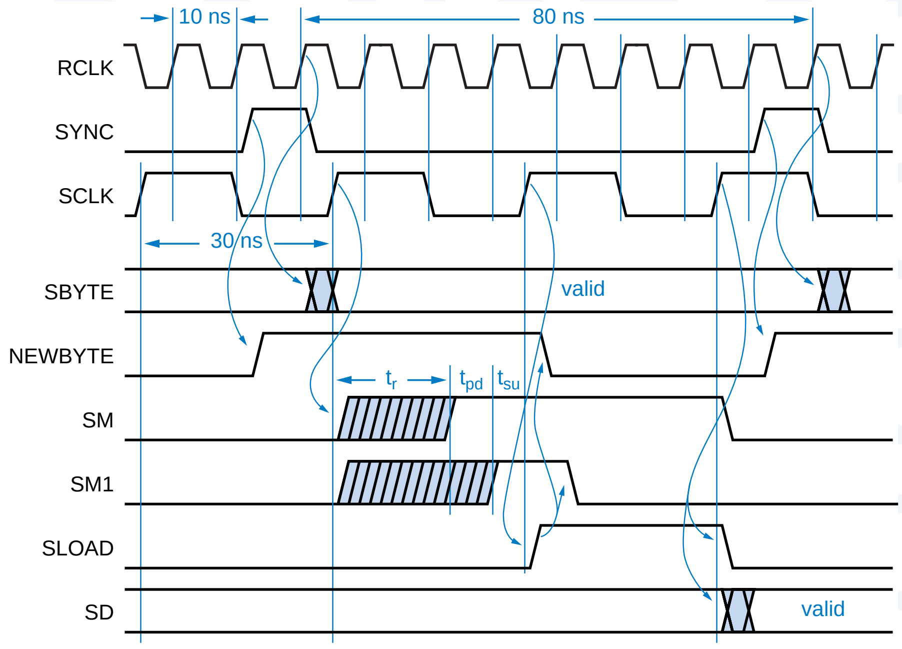


Figure 8-107 Timing for the SCTRL circuit in Figure 8-106.

Since the latch is set on the leading edge of SYNC, we actually cheated a little; NEWBYTE may be asserted a little *before* a new byte is actually available in HREG. This is OK, because we know that it takes two SCLK periods from when NEWBYTE is sampled until SREG is loaded. In fact, we might have cheated even more if an earlier version of SYNC was available (see Exercise 8.76).

Assuming that t_{su} is the setup time of a D flip-flop and t_{pd} is the propagation delay of the AND gate in Figure 8-106, the available metastability resolution time t_r is one SCLK period, 30 ns, minus $t_{su} + t_{pd}$, as shown in Figure 8-107. The timing diagram also shows why we can't use SM directly as the reset signal for the S-R latch. Since SM can be metastable, it could wreak havoc. For example, it could be semi-HIGH long enough to reset the latch but then fall back to LOW; in that case, SLOAD would not get set and we would miss a byte. By using instead the output of the synchronizer (SLOAD) both for the latch reset and for the load signal in the SCLK domain, we ensure that the new byte is detected and handled consistently in both clock domains.

The timing that we showed in Figure 8-107 is nominal, but we also have to analyze what happens if SCLK has a different phase relationship with RCLK and SYNC than what is shown. You should be able to convince yourself that if the SCLK edge occurs earlier, so that it sample NEWBYTE just as it's going HIGH, everything still works as before, and the data transfer just finishes a little sooner. The more interesting case is when SCLK occurs later, so that it just misses NEWBYTE as it's going HIGH, and catches it one SCLK period later. This timing is shown in Figure 8-108.

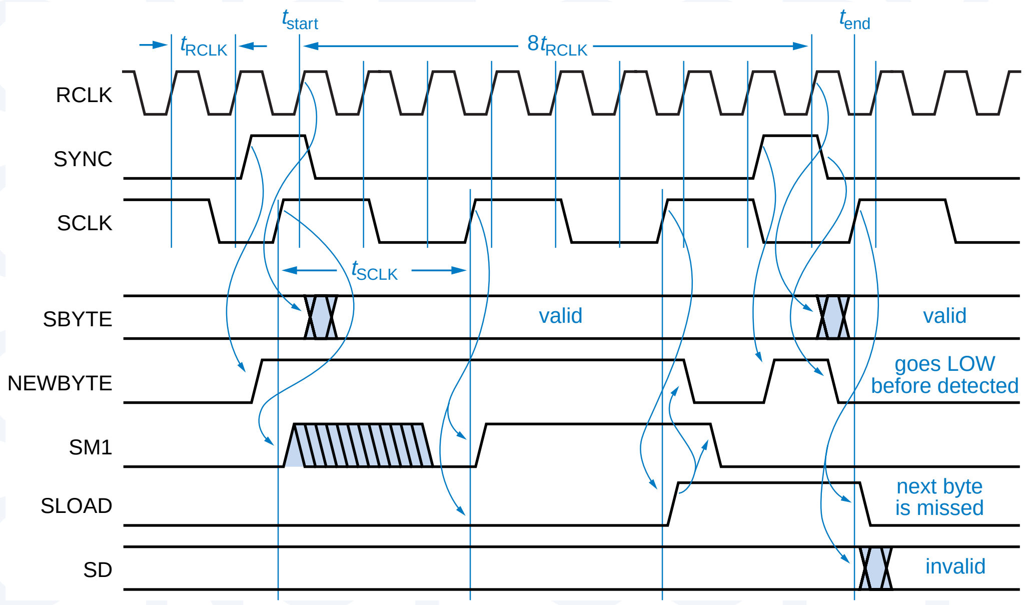


Figure 8-108 Maximum-delay timing for SCTRL circuit.

In the timing diagram, we have shown NEWBYTE going high around the same time as the SCLK edge—less than FF1's t_{su} before the edge. Thus, FF1 may not see NEWBYTE as HIGH or its output may become metastable, and it does not solidly capture NEWBYTE until one SCLK period later. Two SCLK periods after that, we get the SCLK edge that loads SBYTE into SREG.

This timing scenario is bad news, because by the time the load occurs, SBYTE is already changing to the *next* received byte. In addition, SLOAD happens to be asserted during and a little bit after the SYNC pulse for this next received byte. Thus, the latch has both S and R asserted simultaneously. If they are removed simultaneously, the latch output may become metastable. Or, as we've shown in the timing diagram, if NEWBYTE (R) is negated last, then the latch is left in the reset state, and this next received byte is never detected and loaded into the SCLK domain.

Thus, we need to analyze the maximum-delay timing case carefully to determine if a synchronizer will work properly. Figure 8-108 shows a starting reference point t_{start} for the SCTRL circuit, namely the RCLK edge on which a byte is loaded into HREG, at end of SYNC pulse). The ending reference point t_{end} is the SCLK edge on which SBYTE is loaded into SREG. The maximum delay between these two reference points, which we'll call t_{maxd}, is the sum of the following components:

$-t_{RCLK}$ Minus one RCLK period, the delay from t_{start} back to the edge on which SYNC was asserted. This number is negative because SYNC is asserted one clock tick before the tick that actually loads HREG.

t_{CQ}	One flip-flop CLK-to-Q maximum delay. Assuming that SYNC is a direct flip-flop output in the RCLK domain, this is delay from the RCLK edge until SYNC is asserted.
t_{SQ}	Maximum delay from S to Q in the S-R latch in Figure 8-106. This is the delay for NEWBYTE to be asserted.
t_{su}	Setup time of FF1 in Figure 8-106. NEWBYTE must be asserted at or before the setup time to guarantee detection.
t_{SCLK}	One SCLK period. Since RCLK and SCLK are asynchronous, there may be a delay of up to one SCLK period before the next SCLK edge comes along to sample NEWBYTE.
t_{SCLK}	After NEWBYTE is detected by FF1, SLOAD is asserted on the next SCLK tick.
t_{SCLK}	After SLOAD is asserted, SBYTE is loaded into SREG on the next SCLK tick.

Thus, $t_{maxd} = 3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} - t_{RCLK}$. A few other parameters must be defined to complete the analysis:

t_h	The hold time of SREG.
$t_{CQ(min)}$	The minimum CLK-to-Q delay of HREG, conservatively assumed to be 0.
t_{rec}	The recovery time of the S-R latch, the minimum time allowed between negating S and negating R (see box on page 441).

To be loaded successfully into SREG, SBYTE must be remain valid until at least time $t_{end} + t_h$. The point at which SBYTE changes and becomes invalid is 8 RCLK periods after t_{start}, plus $t_{CQ(min)}$. Thus, for proper circuit operation we must have

$$t_{end} + t_h \le t_{start} + 8t_{RCLK}$$

For the maximum-delay case, we substitute $t_{end} = t_{start} + t_{maxd}$ into this relation and subtract t_{start} from both sides to obtain

$$t_{maxd} + t_h \le 8t_{RCLK}$$

Substituting the value of t_{maxd} and rearranging, we obtain

$$3t_{SCLK} + t_{CQ} + t_{SQ} + t_{su} + t_h \le 9t_{RCLK} \qquad (8\text{-}1)$$

as the requirement for correct circuit operation. Too bad. Even if we assume very short component delays (t_{CQ}, t_{SQ}, t_{su}, t_h), we know that $3t_{SCLK}$ (90 ns) plus anything is going to be more than $9t_{RCLK}$ (also 90 ns). So this design will never work properly in the maximum-delay case.

Even if the load-delay analysis gave a good result, we would still have to consider the requirements for proper operation of the SCTRL circuit itself. In particular, we must ensure that when the SYNC pulse for the next byte occurs, it is not negated until time t_{rec} after SLOAD for the previous byte was negated. So, another condition for proper operation is

$$t_{\mathrm{end}} + t_{\mathrm{CQ}} + t_{\mathrm{rec}} \le t_{\mathrm{start}} + 8t_{\mathrm{RCLK}} + t_{\mathrm{CQ(min)}}$$

Substituting and simplifying as before, we get another requirement that isn't met by our design:

$$3t_{\mathrm{SCLK}} + 2t_{\mathrm{CQ}} + t_{\mathrm{SQ}} + t_{\mathrm{su}} + t_{\mathrm{rec}} \le 9\,t_{\mathrm{RCLK}} \qquad (8\text{-}2)$$

There are several ways that we can modify our design to satisfy the worst-case timing requirements. Early in our discussion, we noted that we "cheated" by asserting SYNC one RCLK period before the data in HREG is valid, and that we actually might get away with asserting SYNC even soon. Doing this can help us meet the maximum delay requirement, because it reduces the "$8t_{\mathrm{RCLK}}$" term on the right-hand side of the relations. For example, if we asserted SYNC two RCLK periods earlier, we would reduce this term to "$6t_{\mathrm{RCLK}}$". However, there's no free lunch, we can't assert SYNC arbitrarily early. We must also consider a *minimum delay* case, to ensure that the new byte is actually available in HREG when SBYTE is loaded into SREG. The minimum delay t_{maxd} between t_{start} and t_{end} is the sum of the following components:

$-nt_{\mathrm{RCLK}}$	Minus n RCLK periods, the delay from t_{start} back to the edge on which SYNC was asserted. In the original design, $n = 1$.
$t_{\mathrm{CQ(min)}}$	This is the minimum delay from the RCLK edge until SYNC is asserted, conservatively assumed to be 0.
t_{SQ}	This is the delay for NEWBYTE to be asserted, again assumed to be 0.
$-t_{\mathrm{h}}$	Minus the hold time of FF1 in Figure 8-106. NEWBYTE might be asserted at the end of the hold time and still be detected.
$0t_{\mathrm{SCLK}}$	Zero times the SCLK period. We might get "lucky" and have the SCLK edge come along just as the hold time of FF1 is ending.
t_{SCLK}	A one-SCLK-period delay to asserting SLOAD, as before.
t_{SCLK}	A one-SCLK-period delay to loading SBYTE into SREG, as before.

In other words, $t_{\mathrm{mind}} = 2t_{\mathrm{SCLK}} - t_{\mathrm{h}} - nt_{\mathrm{RCLK}}$.

For this case, we must ensure that the new byte has propagated to the output of HREG when the setup time window of SREG begins, so we must have

$$t_{\mathrm{end}} - t_{\mathrm{su}} \ge t_{\mathrm{start}} + t_{\mathrm{co}},$$

where t_{co} is the maximum clock-to-output delay of HREG. Substituting $t_{\mathrm{end}} = t_{\mathrm{start}} + t_{\mathrm{mind}}$ and subtracting t_{start} from both sides, we get

$$t_{\mathrm{mind}} - t_{\mathrm{su}} \ge t_{\mathrm{co}}.$$

Substituting the value of t_{mind} and rearranging, we get the final requirement,

$$2t_{\mathrm{SCLK}} - t_{\mathrm{h}} - t_{\mathrm{su}} - t_{\mathrm{co}} \ge nt_{\mathrm{RCLK}} \qquad (8\text{-}3)$$

If, for example, t_h, t_{su}, and t_{co} are 10 ns each, the maximum value of n is 3; we can't generate SYNC more than two clock ticks before its original position in Figure 8-108. This may or may not be enough to solve the maximum-delay problem, depending on other delay values; this is explored for a particular set of components in Exercise 8.76.

Moving the SYNC pulse earlier may not give enough delay improvement or may not be an available option in some systems. An alternative solution that can always be made to work is to increasing the time between successive data transfers from one clock domain to the other. We can always do this because we can always transfer more bits per synchronization. In the Ethernet-interface example, we could collect 16 bits at a time in the RCLK domain and transfer 16 bits at a time to the SCLK domain. This changes the previously stated $8t_{\text{RCLK}}$ terms to $16t_{\text{RCLK}}$, providing a lot more margin for the maximum-delay timing requirements. Once 16 bits have been transferred into the SCLK domain, we can still break them into two 8-bit chunks if we need to process the data a byte at a time.

It may also be possible to improve performance by modifying the design of the SCTRL circuit. Figure 8-111 shows a version where SLOAD is generated directly by the flip-flop that samples NEWBYTE. In this way, SLOAD appears one SCLK period sooner than in our original SCTRL circuit. Also, the S-R latch is cleared sooner. This circuit works only if a couple of key assumptions are true:

1. A reduced metastability resolution time for FF1 is acceptable, equal to the time that SCLK is HIGH. Metastability must be resolved before SCLK goes LOW, because that's when the S-R latch gets cleared if SLOAD is HIGH.
2. The setup time of SREG's CLKEN input (Figure 8-102) is less than or equal to the time that SCLK is LOW. Under the previous assumption, the SLOAD signal applied to CLKEN might be metastable until SCLK goes LOW.
3. The time that SCLK is LOW is long enough to generate a reset pulse on RNEW that meets the minimum pulse-width requirement of the S-R latch.

Note that these behaviors makes proper circuit operation dependent on the duty cycle of SCLK. If SCLK is relatively slow and its duty cycle is close to 50%, this circuit generally works fine. But if SCLK is too fast or has a very small, very large, or unpredictable duty cycle, the original circuit approach must be used.

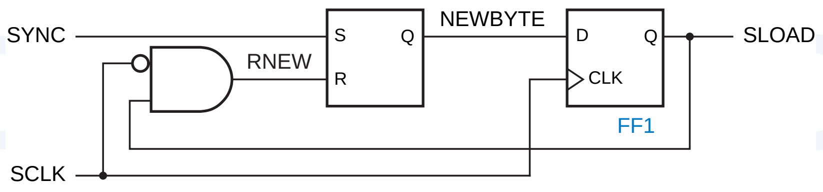


Figure 8-109 Half-clock-period SCTRL circuit for generating SLOAD.

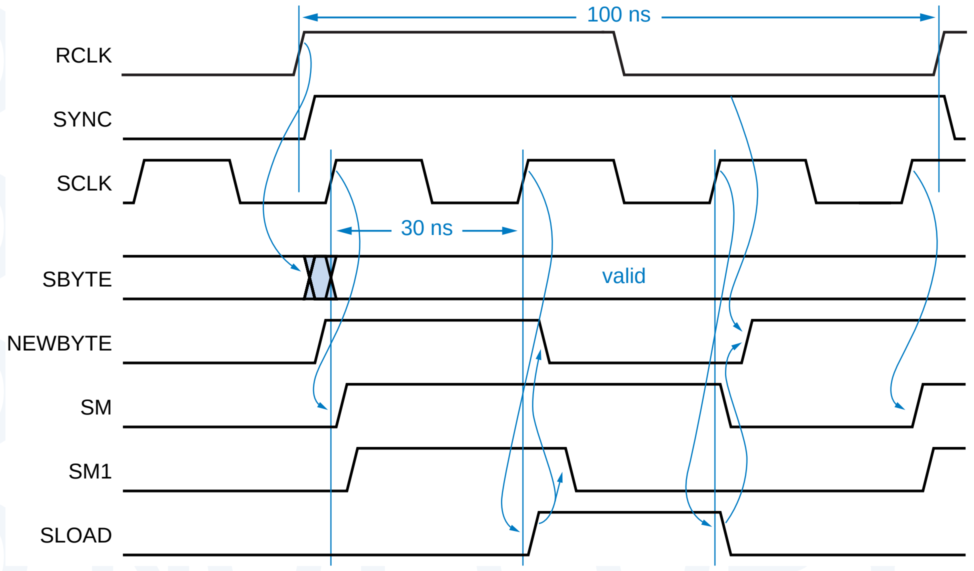
All of these synchronization schemes require the clock frequencies to be within a certain range of each other for proper circuit operation. This must be considered for testing, where the clocks are usually run slower, and for upgrades, where one or both clocks may run faster. For example, in the Ethernet interface example, we wouldn't change the frequency of standard 100-Mbps Ethernet, but we might upgrade the PCI bus from 33 to 66 MHz.

The problems caused by clock frequency changes can be subtle. To get a better handle on what can go wrong, it's useful to consider how a synchronizer works (or doesn't work!) if one clock frequency changes by a factor of 10 or more.

For example what happens to the synchronizer timing in Figure 8-107 if we change RCLK from 100 MHz to 10 MHz? At first glance, it would seem that all is well, since a byte now arrives only once every 800 ns, giving much more time for the byte to be transferred into the SCLK domain. Certainly, Eqn. 8-1 on page 681 and Eqn. 8-2 on page 682 are satisfied with much more margin. However, Eqn. 8-3 is no longer satisfied unless we reduce *n* to zero! This could be accomplished by generating SYNC one RCLK tick later than is shown in Figure 8-107.

But even with this change, there's *still* a problem. Figure 8-110 shows the new timing, including the later SYNC pulse. The problem is that the SYNC pulse is now 100 ns long. As before, NEWBYTE (the output of the S-R latch in

Figure 8-110 Synchronizer timing with slow (10 MHz) RCLK.



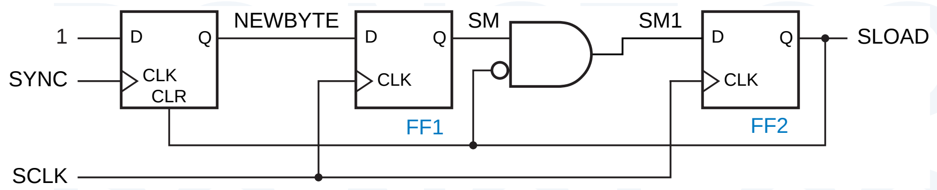


Figure 8-111 Synchronizer with edge-triggered SYNC detection.

Figure 8-106 on page 678) is set by SYNC and is cleared by SLOAD. The problem is that when SLOAD goes away, SYNC is still asserted, as shown in the new timing diagram. Thus, the new byte will be detected and transferred twice!

The solution to the problem is to detect only the leading edge of SYNC, so that the circuit is not sensitive to the length of the SYNC pulse. A common way of doing this is to replace the S-R latch with an edge-triggered D flip-flop, as shown in Figure 8-111. The leading edge of SYNC sets the flip-flop, while SLOAD is used as an asynchronous clear as before.

The circuit in Figure 8-111 solves the slow-RCLK problem, but it also changes the derivation of Eqns. 8-1 through 8-3 and may make timing more constrained in some areas (see Exercise 8.77). Another disadvantage that this circuit cannot be realized in a typical PLD, which has all flip-flops controlled by the same clock; instead, a discrete flip-flop must be used to detect SYNC.

After reading almost 10 pages to analyze just one “simple” example, you should have a strong appreciation of the difficulty of correct synchronization-circuit design. Several guidelines can help you:

- Minimize the number of different clock domains in a system.
- Clearly identify all clock boundaries and provide clearly identified synchronizers at those boundaries.
- Provide sufficient metastability resolution time for each synchronizer so that synchronizer failure is rare, much more unlikely than other hardware failures.
- Analyze synchronizer behavior over a range of timing scenarios, including faster and slower clocks that might be applied as a result of system testing or upgrades.
- Simulate system behavior over a wide range of timing scenarios as well.

The last guideline above is a catch-all for modern digital designers, who usually rely on sophisticated, high-speed logic simulators to find their bugs. But it’s not a substitute for following the first four guidelines. Ignoring them can lead to problems that cannot be detected by a typical, small number of simulation scenarios. Of all digital circuits, synchronizers are the ones for which it’s most important to be “correct by design”!

References

Probably the first comprehensive set of examples of sequential MSI parts and applications appeared in *The TTL Applications Handbook*, edited by Peter Alfke and Ib Larsen (Fairchild Semiconductor, 1973). This highly practical and informative and book was invaluable to this author and to many others who developed digital design curricula in the 1970s.

Another book that emphasizes design with larger-scale combinational and sequential logic functions is Thomas R. Blakeslee's *Digital Design with Standard MSI and LSI*, 2nd ed., (Wiley, 1979). Blakeslee's coverage of the concept of space/time trade-offs is excellent, and he also one of the first to introduce the microprocessor as "a universal logic circuit."

Moving quickly from the almost forgotten to the yet-to-be discovered, manufacturers' web sites are an excellent source of information on digital design practices. For example, a comprehensive discussion of bus hold circuits can be found in Texas Instruments' "Implications of Slow or Floating CMOS Inputs" (publ. SCBA004B, December 1997), available on TI's web site at `www.ti.com`. Another discussion appears in Fairchild Semiconductor's "Designing with Bushold" (*sic*, publ. AN-5006, April 1999), at `www.fairchildsemi.com`.

Announcements and data sheets for all kinds of new, improved MSI and larger parts can also be found on the web. Following a certain automobile manufacturer's proclamation in the 60s and then again in the late 90s that "wider is better," logic manufacturers have also introduced "wide-bus" registers, drivers, and transceivers that cram 16, 18, or even 32 bits of function into a high pin-count surface-mount package. Descriptions of many such parts can be found at the Texas Instruments web site (search for "widebus"). Other sites with a variety of logic data sheets, application notes, and other information include Motorola (`www.mot.com`), Fairchild Semiconductor (`www.fairchildsemi.com`), and Philips Semiconductor (`www.philipslogic.com`).

The field of logic design is fast moving, so much so that sometimes I wish that I wrote fiction, so that I wouldn't have to revise the "practices" discussions in this book every few years. Lucky for me, this book does cover some unchanging theoretical topics (a.k.a. "principles"), and this chapter is no exception. Logic hazards have been known since at least the 1950s, and the function hazards were discussed by Edward J. McCluskey in *Logic Design Principles* (Prentice Hall, 1986). Galois fields were invented centuries ago, and their applications to error-correcting codes, as well as to the LFSR counters of this chapter, are described in introductory books on coding theory, including *Error-Control Techniques for Digital Communication* by A. M. Michelson and A. H. Levesque (Wiley-Interscience, 1985). A mathematical theory of state-machine decomposition has been studied for years; Zvi Kohavi devotes a chapter to the topic in his classic book *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill, 1978). Bur let us now return to the less esoteric.

As we mentioned in Section 8.4.5, some PLDs and CPLDs contain XOR structures that allow large counters to be designed without a large number of product terms. This requires a somewhat deeper understanding of counter excitation equations, as described in Section 10.5 of the second edition of this book. This material can also be found on the web at `www.ddpp.com`.

The general topics of clock skew and multiphase clocking are discussed in McCluskey's *Logic Design Principles*, while an illuminating discussion of these topics as applied to VLSI circuits can be found in *Introduction to VLSI Systems* by Carver Mead and Lynn Conway (Addison-Wesley, 1980). Mead and Conway also provide an introduction to the important topic of *self-timed systems* that eliminate the system clock, allowing each logic element to proceed at its own rate. To give credit where credit is due, we note that all of the Mead and Conway material on system timing, including an outstanding discussion of metastability, appears in their Chapter 7 written by Charles L. Seitz.

self-timed systems

Thomas J. Chaney spent decades studying and reporting on the metastability problem. One of his more important works, "Measured Flip-Flop Responses to Marginal Triggering" (*IEEE Trans. Comput.*, Vol. C-32, No. 12, December 1983, pp. 1207–1209, reports some of the results that we showed in Table 8-35.

For the mathematically inclined, Lindsay Kleeman and Antonio Cantoni have written "On the Unavoidability of Metastable Behavior in Digital Systems" (*IEEE Trans. Comput.*, Vol. C-36, No. 1, January 1987, pp. 109–112); the title says it all. The same authors posed the question, "Can Redundancy and Masking Improve the Performance of Synchronizers?" (*IEEE Trans. Comput.*, Vol. C-35, No. 7, July 1986, pp. 643–646). Their answer in that paper was "no," but a response from a reviewer caused them to change their minds to "maybe." Obviously, they've gone metastable themselves! (Having two authors and a reviewer hasn't improved their performance, so the obvious answer to their original question is "no!") In any case, Kleeman and Antonio's papers provide a good set of pointers to mainstream scholarly references on metastability.

The most comprehensive set of early references on metastability (not including Greek philosophers or Devo) is Martin Bolton's "A Guided Tour of 35 Years of Metastability Research" (*Proc. Wescon 1987*, Session 16, "Everything You Might Be Afraid to Know about Metastability," Wescon Session Records, `www.wescon.com`, 8110 Airport Blvd., Los Angeles, CA 90045).

In recent years, as system clock speeds have steadily increased, many IC manufacturers have become much more active in studying and publishing the metastability characteristics of their devices on the web. Texas Instruments (`www.ti.com`) provides a very good discussion including test circuits and measured parameters for ten different logic families in "Metastable Response in 5-V Logic Circuits" by Eilhard Haseloff (TI pub. SDYA006, 1997). Cypress Semiconductor (`www.cypress.com`) published an application note, "Are Your PLDs Metastable?" (1997) that is an excellent reference including some history (going back to 1952!), an analog circuit analysis of metastability, test and measurement circuits, and metastability parameters for Cypress PLDs. Another recent note is “Metastability Considerations” from Xilinx Corporation (`www.xilinx.com`, publ. XAPP077, 1997), which gives measured parameters for their XC7300 and XC9500 families of CPLDs. Of particular interest is the clever circuit and methodology that allows them to count metastable events *inside* the device, even though metastable waveforms are not observable on external pins.

Most digital design textbooks now give good coverage to metastability, prompted by the existence of metastability in real circuits and perhaps also by competition—since 1990, this textbook has been hammering on the topic by introducing metastability in its earliest coverage of sequential circuits. On the analog side of the house, Howard Johnson and Martin Graham provide a nice introduction and a description of how to observe metastable states in their *High-Speed Digital Design: A Handbook of Black Magic* (Prentice Hall, 1993).

Drill Problems

8.1 Compare the propagation delays from AVALID to a chip-select output for the two decoding approaches shown in Figures 8-15 and 8-16. Assume that 74FCT373 latches and 10-ns GAL16V8C devices are used in both designs. Repeat for the delay from ABUS to a chip-select output.

8.2 Suppose that in Table 8-2, the second RAM bank (`RAMCS1`) is decoded instead using the expression `((ABUS >= 0x010) & (ABUS < 0x020))`. Does this yield the same results as the original expression, `(ABUS == RAMBANK0)`? Explain.

8.3 What would happen if you replaced the edge-triggered D flip-flops in Figure 7-38 with D latches?

8.4 What is the counting sequence of the circuit shown in Figure X8.4?

8.5 What is the behavior of the counter circuit of Figure 8-36 if it is built using a 74x161 instead of a 74x163?

8.6 A 74x163 counter is hooked up with inputs ENP, ENT, A, and D always HIGH, inputs B and C always LOW, input LD_L = (QB · QC)′, and input CLR_L = (QC · QD)′. The CLK input is hooked up to a free-running clock signal. Draw a logic diagram for this circuit. Assuming that the counter starts in state 0000, write the output sequence on QD QC QB QA for the next 15 clock ticks.

8.7 Determine the widths of the glitches shown in Figure 8-43 on the Y2_L output of a 74x138 decoder, assuming that the ’138 is internally structured as shown in Figure 5-37(a) on page 320, and that each internal gate has a delay of 10 ns.

8.8 Calculate the minimum clock period of the data unit in Figure 8-82. Use the maximum propagation delays given in Table 5-3 for LS-TTL combinational parts. Unless you can get the real numbers from a TTL data book, assume that all registers require a 10 ns minimum setup time on inputs and have a 20 ns propagation delay from clock to outputs. Indicate any assumptions you’ve made about delays in the control unit.

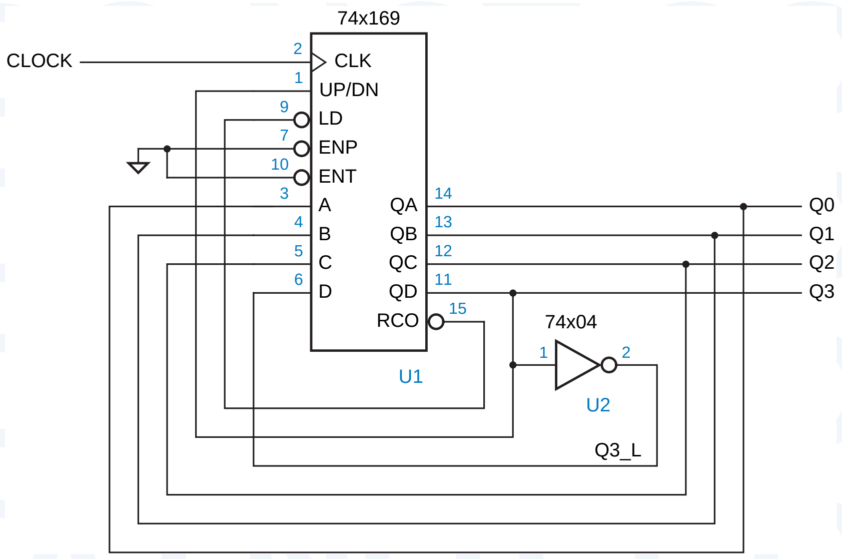


Figure X8.4

8.9 Calculate the MTBF of a synchronizer built according to Figure 8-96 using 74F74s, assuming a clock frequency of 25 MHz and an asynchronous transition rate of 1 MHz. Assume that the setup time of an 'F74 is 5 ns and the hold time is zero.

8.10 Calculate the MTBF of the synchronizer shown in Figure X8.10, assuming a clock frequency of 25 MHz and an asynchronous transition rate of 1 MHz. Assume that the setup time t_{setup} and the propagation delay t_{pd} from clock to Q or QN in a 74ALS74 are both 10 ns.

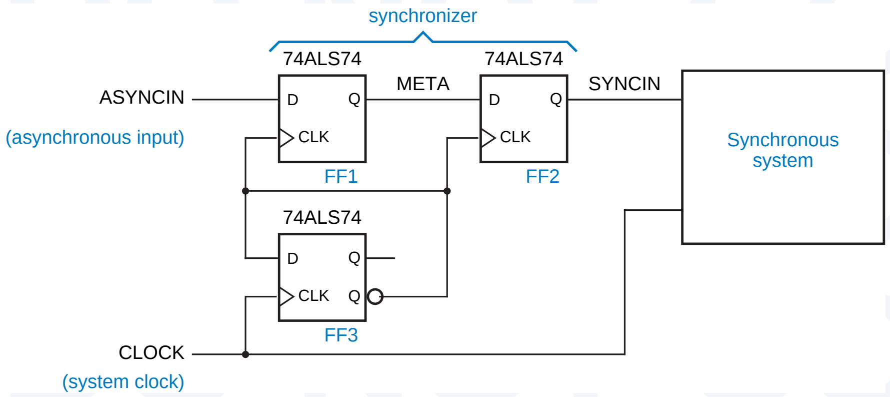


Figure X8.10

Exercises

8.11 What does the TTL Data Book have to say about momentarily shorting the outputs of a gate to ground as we do in the switch debounce circuit of Figure 8-5?

8.12 Investigate the behavior of the switch debounce circuit of Figure 8-5 if 74HCT04 inverters are used; repeat for 74AC04 inverters.

8.13 Suppose you are asked to design a circuit that produces a debounced logic input from an SPST (single-pole, *single*-throw) switch. What inherent problem are you faced with?

8.14 Explain why CMOS bus holder circuits don’t work well on three-state buses with TTL devices attached. (*Hint:* Consider TTL input characteristics.)

8.15 Write a single VHDL program that combines the address latch and latching decoder of Figure 8-16 and Table 8-2. Use the signal name LA[19:0] for the latched address outputs.

8.16 Design a 4-bit ripple counter using four D flip-flops and no other components.

8.17 What is the maximum propagation delay from clock to output for the 4-bit ripple counter of Exercise 8.16 using 74HCT flip-flops? Repeat, using 74AHCT and 74LS74 flip-flops.

8.18 Design a 4-bit ripple *down* counter using four D flip-flops and no other components.

8.19 What limits the maximum counting speed of a ripple counter, if you don’t insist on being able to read the counter value at all times?

8.20 Based on the design approach in Exercise 8.16 and the answer to Exercise 8.19, what is the maximum counting speed (frequency) for a 4-bit ripple counter using 74HCT flip-flops? Repeat, using 74AHCT and 74LS74 flip-flops.

8.21 Write a formula for the maximum clock frequency of the synchronous serial binary counter circuit in Figure 8-28. In your formula, let t_{TQ} denote the propagation delay from T to Q in a T flip-flop, t_{setup} the setup time of the EN input to the rising edge of T, and t_{AND} the delay of an AND gate.

8.22 Repeat Exercise 8.21 for the synchronous parallel binary counter circuit shown in Figure 8-29, and compare results.

8.23 Repeat Exercise 8.21 for an n-bit synchronous serial binary counter.

8.24 Repeat Exercise 8.21 for an n-bit synchronous parallel binary counter. Beyond what value of n is your formula no longer valid?

8.25 Using a 74x163 4-bit binary counter, design a modulo-11 counter circuit with the counting sequence 3, 4, 5, …, 12, 13, 3, 4, ….

8.26 Look up the internal logic diagram for a 74x162 synchronous decade counter in a data book, and write its state table in the style of Table 8-11, including its counting behavior in the normally unused states 10–15.

8.27 Devise a cascading scheme for 74x163s, analogous to the synchronous parallel counter structure of Figure 8-29, such that the maximum counting speed is the same for any counter with up to 36 bits (nine ’163s). Determine the maximum counting speed using worst-case delays from a manufacturer’s data book for the ’163s and any SSI components used for cascading.

8.28 Design a modulo-129 counter using two 74x163s and a single inverter.

8.29 Design a clocked synchronous circuit with four inputs, N3, N2, N1, and N0, that represent an integer N in the range 0–15. The circuit has a single output Z that is asserted for exactly N clock ticks during any 16-tick interval (assuming that N is held constant during the interval of observation). (*Hints:* Use combinational logic with a 74x163 set up as a free-running divide-by-16 counter. The ticks in which

Z is asserted should be spaced as evenly as possible, that is, every second tick when $N = 8$, every fourth when $N = 4$, and so on.)

8.30 Modify the circuit of Exercise 8.29 so that Z produces *N transitions* in each 16-tick interval. The resulting circuit is called a *binary rate multiplier,* and was once sold as a TTL MSI part, the 7497. (*Hint:* Gate the clock with the level output of the previous circuit.)

binary rate multiplier

8.31 A digital designer (the author!) was asked at the last minute to add new functionality to a PCB that had room for just one more 16-pin IC. The PCB already had a 16-MHz clock signal, MCLK, and a spare microprocessor-controlled select signal, SEL. The designer was asked to provide a new clock signal, UCLK, whose frequency would be 8 MHz or 4 MHz depending on the value of SEL. To make things worse, the PCB had no spare SSI gates, and UCLK was required to have a 50% duty cycle at both frequencies. It took the designer about five minutes to come up with a circuit. Now it's your turn to do the same. (*Hint:* The designer had long considered the 74x163 to be the fundamental building block of tricky sequential-circuit design.)

8.32 Design a modulo-16 counter, using one 74x169 and at most one SSI package, with the following counting sequence: 7, 6, 5, 4, 3, 2, 1, 0, 8, 9, 10, 11, 12, 13, 14, 15, 7, ….

8.33 Modify the VHDL program in Table 8-14 so that the type of ports `D` and `Q` is `STD_LOGIC_VECTOR`, including conversion functions as required.

8.34 Modify the program in Table 8-16 to use structural VHDL, so it conforms exactly to the circuit in Figure 8-45, including the signal names shown in the figure. Define and use any of the following entities that don't already exist in your VHDL library: `AND2`, `INV`, `NOR2`, `OR2`, `XNOR2`, `Vdffqqn`.

8.35 Modify the program in Table 8-17 to use VHDL's `generic` statement, so that the counter size can be changed using the `generic` definition.

8.36 Design a parallel-to-serial conversion circuit with eight 2.048 Mbps, 32-channel serial links and a single 2.048 MHz, 8-bit, parallel data bus that carries 256 bytes per frame. Each serial link should have the frame format defined in Figure 8-55. Each serial data line SDATAi should have its own sync signal SYNCi; the sync pulses should be staggered so that SYNCi + 1 has a pulse one clock tick after SYNCi.

Show the timing of the parallel bus and the serial links, and write a table or formula that shows which parallel-bus timeslots are transmitted on which serial links and timeslots. Draw a logic diagram for the circuit using MSI parts from this chapter; you may abbreviate repeated elements (e.g., shift registers), showing only the unique connections to each one.

8.37 Repeat Exercise 8.36, assuming that all serial data lines must reference their data to a single, common SYNC signal. How many more chips does this design require?

8.38 Show how to enhance the serial-to-parallel circuit of Exercise 8-57 so that the byte received in each timeslot is stored in its own register for 125 μs, until the next byte from that timeslot is received. Draw the counter and decoding logic for 32 timeslots in detail, as well as the parallel data registers and connections for timeslots 31, 0, and 1. Also draw a timing diagram in the style of Figure 8-58 that shows the decoding and data signals associated with timeslots 31, 0, and 1.

8.39 Suppose you are asked to design a serial computer, one that moves and processes data one bit at a time. One of the first decisions you must make is which bit to transmit and process first, the LSB or the MSB. Which would you choose, and why?

8.40 Design an 8-bit self-correcting ring counter whose states are 11111110, 11111101, …, 01111111, using only two SSI/MSI packages.

8.41 Design two different 2-bit, 4-state counters, where each design uses just a single 74x74 package (two edge-triggered D flip-flops) and no other gates.

8.42 Design a 4-bit Johnson counter and decoding for all eight states using just three SSI/MSI packages. Your counter need not be self-correcting.

8.43 Starting with state 0001, write the sequence of states for a 4-bit LFSR counter designed according to Figure 8-68 and Table 8-21.

8.44 Prove that an even number of shift-register outputs must be connected to the odd-parity circuit in an n-bit LFSR counter if it generates a maximum-length sequence. (Note that this is a necessary but not a sufficient requirement. Also, although Table 8-21 is consistent with what you're supposed to prove, simply quoting the table is not a proof!)

8.45 Prove that X0 must appear on the right-hand side of any LFSR feedback equation that generates a maximum-length sequence. (*Note:* Assume the LFSR bit ordering and shift direction are as given in the text; that is, the LFSR counter shifts right, toward the X0 stage.)

8.46 Suppose that an n-bit LFSR counter is designed according to Figure 8-68 and Table 8-21. Prove that if the odd-parity circuit is changed to an even-parity circuit, the resulting circuit is a counter that visits $2^n - 1$ states, including all of the states except 11…11.

8.47 Find a feedback equation for a 3-bit LFSR counter, other than the one given in Table 8-21, that produces a maximum-length sequence.

8.48 Given an n-bit LFSR counter that generates a maximum-length sequence ($2^n - 1$ states), prove that an extra XOR gate and an $n - 1$ input NOR gate connected as suggested in Figure 8-69 produces a counter with 2^n states.

8.49 Prove that a sequence of 2^n states is still obtained if a NAND gate is substituted for a NOR above, but that the state sequence is different.

8.50 Design an iterative circuit for checking the parity of a 16-bit data word with a single even parity bit. Does the order of bit transmission matter?

8.51 Modify the shift-register program in Table 8-23 to provide an asynchronous clear input using a 22V10.

8.52 Write an ABEL program that provides the same functionality as a 74x299 shift register. Show how to fit this function into a single 22V10 or explain why it does not fit.

8.53 In what situations do the ABEL programs in Tables 8-26 and 8-27 give different operational results?

8.54 Modify the ABEL program in Table 8-26 so that the phases are always at least two clock ticks long, even if RESTART is asserted at the beginning of a phase. RESET should still take effect immediately.

8.55 Repeat the preceding exercise for the program in Table 8-27.

8.56 A student proposed to create the timing waveforms of Figure 8-72 by starting with the ABEL program in Table 8-27 and changing the encoding of each of states P1F, P2F, . . . , P6F so that the corresponding phase output is 1 instead of 0, so that the phase output is 0 only during the second tick of each phase, as required. Is this a good approach? Comment on the results produced by the ABEL compiler produce when you try this.

8.57 The outputs waveforms produced by the ABEL programs in Tables 8-29 and 8-30 are not identical when the RESTART and RUN inputs are changed. Explain the reason for this, and then modify the program in Table 8-30 so that its behavior matches that of Table 8-29.

8.58 The ABEL ring-counter implementation in Table 8-26 is not self-synchronizing. For example, describe what happens if the outputs [P1_L..P6_L] are initially all 0, and the RUN input is asserted without ever asserting RESET or RESTART. What other starting states exhibit this kind of non-self-synchronizing behavior? Modify the program so that it *is* self-synchronizing.

8.59 Repeat the preceding exercise for the VHDL ring-counter implementation in Table 8-33.

8.60 Design an iterative circuit with one input B_i per stage and two boundary outputs X and Y such that X = 1 if at least two B_i inputs are 1 and Y = 1 if at least two *consecutive* B_i inputs are 1.

8.61 Design a combination-lock machine according to the state table of Table 7-14 on page 486 using a single 74x163 counter and combinational logic for the LD_L, CLR_L, and A–D inputs of the '163. Use counter values 0–7 for states A–H.

8.62 Write an ABEL program corresponding to the state diagram in Figure 8-84 for the multiplier control unit.

8.63 Write a VHDL program corresponding to the state diagram in Figure 8-84 for the multiplier control unit.

8.64 Write a VHDL program that performs with the same inputs, outputs, and functions as the multiplier data unit in Figure 8-82.

8.65 Write a VHDL program that combines the programs in the two preceding exercises to form a complete 8-bit shift-and-add multiplier.

8.66 The text stated that the designer need not worry about any timing problems in the synchronous design of Figure 8-83. Actually, there is one slight worry. Look at the timing specifications for the 74x377 and discuss.

8.67 Determine the minimum clock period for the shift-and-add multiplier circuit in Figure 8-83, assuming that the state machine is realized in a single GAL16V8-20 and that the MSI parts are all 74LS TTL. Use worst-case timing information from the tables in this book. For the '194, t_{pd} from clock to any output is 26 ns, and t_s is 20 ns for serial and parallel data inputs and 30 ns for mode-control inputs.

8.68 Design a data unit and a control-unit state machine for multiplying 8-bit two's-complement numbers using the algorithm discussed in Section 2.8.

8.69 Design a data unit and control-unit state machine for dividing 8-bit unsigned numbers using the shift-and-subtract algorithm discussed in Section 2.9.

8.70 Suppose that the SYNCIN signal in Drill 8.10 is connected to a combinational circuit in the synchronous system, which in turn drives the D inputs of 74ALS74 flip-flops that are clocked by CLOCK. What is the maximum allowable propagation delay of the combinational logic?

8.71 The circuit in Figure X8.71 includes a deskewing flip-flop so that the synchronized output from the multiple-cycle synchronizer is available as soon as possible after the edge of CLOCK. Ignoring metastability considerations, what is the maximum frequency of CLOCK? Assume that for a 74F74, $t_{\text{setup}} = 5$ ns and $t_{\text{pd}} = 7$ ns.

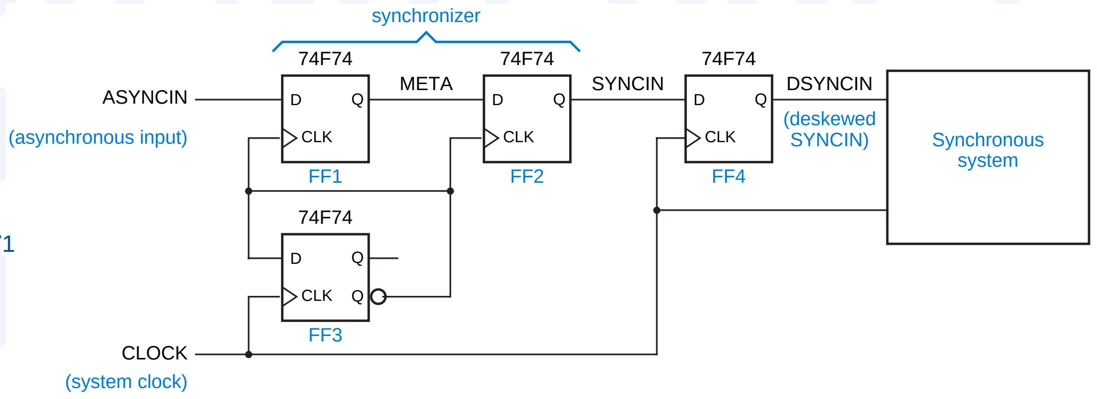
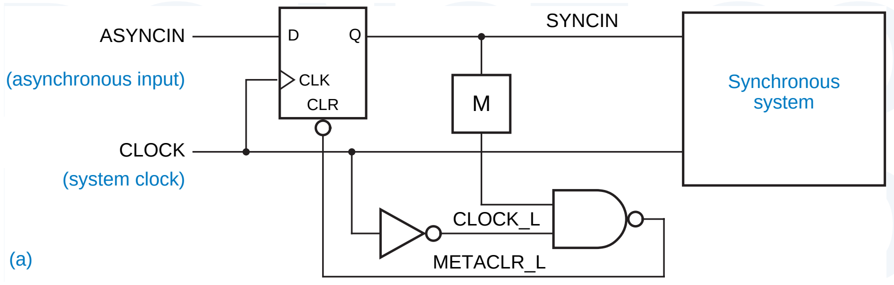


Figure X8.71

8.72 Using the maximum clock frequency determined in Exercise 8.71, and assuming an asynchronous transition rate of 4 MHz, determine the synchronizer's MTBF.

8.73 Determine the MTBF of the synchronizer in Figure X8.71, assuming an asynchronous transition rate of 4 MHz and a clock frequency of 40 MHz, which is less than the maximum determined in Figure X8.71. In this situation, "synchronizer failure" really occurs only if DSYNCIN is metastable. In other words, SYNCIN may be allowed to be metastable for a short time, as long as it doesn't affect DSYNCIN. This yields a better MTBF.

8.74 A famous digital designer devised the circuit shown in Figure X8.74(a), which is supposed to eliminate metastability within one period of a system clock. Circuit M is a memoryless analog voltage detector whose output is 1 if Q is in the metastable state, 0 otherwise. The circuit designer's idea was that if line Q is detected to be in the metastable state when CLOCK goes low, the NAND gate will clear the D flip-flop, which in turn eliminates the metastable output, causing a 0 output from circuit M and thus negating the CLR input of the flip-flop. The circuits are all fast enough that this all happens well before CLOCK goes high again; the expected waveforms are shown in Figure X8.74(b).



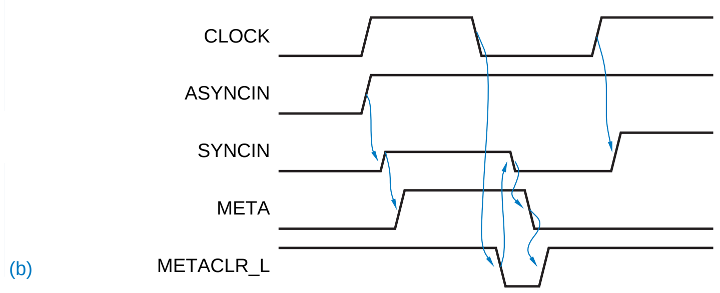


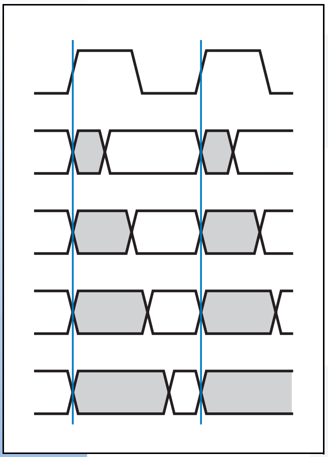
Figure X8.74

Unfortunately, the synchronizer still failed occasionally, and the famous digital designer is now designing pockets for blue jeans. Explain, in detail, how it failed, including a timing diagram.

8.75 Look up U.S. patent number 4,999,528, “Metastable-proof flip-flop,” and describe why it doesn’t always work as advertised. (*Hint:* There’s enough information in the abstract to figure out how it can fail.)

8.76 In the synchronization circuit of Figures 8-102, 8-104, and Figure 8-106, you can reduce the delay of the transfer of a byte from the RCLK domain to the SCLK domain if you use an earlier version of the SYNC pulse to start the synchronizer. Assuming that you can generate SYNC during any bit of the received byte, which bit should you use to minimize the delay? Also determine whether your solution satisfies the maximum-delay requirements for the circuit. Assume that all the components have 74AHCT timing and that the S-R latch is built from a pair of cross-coupled NOR gates, and show a detailed timing analysis for your answers.

8.77 Instead of using a latch in the synchronization control circuit of Figure 8-106, some applications use an edge-triggered D flip-flop as shown in Figure 8-111. Derive the maximum-delay and minimum-delay requirements for this circuit, corresponding to Eqns. 8-1 through 8-3, and discuss whether this approach eases or worsens the delay requirements.



chapter 9

Sequential-Circuit Design Examples

Just about every real digital system is a sequential circuit—all it takes is one feedback loop, latch, or flip-flop to make a circuit's present behavior depend on its past inputs. However, if we were to design or analyze a typical digital system as a single sequential circuit, we would end up with an enormous state table. For example, strictly speaking, a PC with only 16 Mbytes of main memory is a sequential circuit with well over $2^{128,000,000}$ states!

We noted in previous chapters that we typically deal with digital systems in smaller chunks, for example, partitioning them into data paths, registers, and control units (Section 8.7.1). In fact, a typical system has multiple functional units with well defined interfaces and connections between them (as supported by VHDL, Section 4.7.2), and each functional unit may contain a hierarchy with several layers of abstraction (as supported by both VHDL and typical schematic drawing packages [Section 5.1.6]). Thus, we can deal with sequential circuits in much smaller chunks.

After all of this build-up, I have to admit that the design of complex, hierarchical digital systems is beyond the scope of this text. However, the heart of any system or any of its subsystems is typically a state machine or other sequential circuit, and that's something we *can* study here. So, this chapter will try to reinforce your understanding of sequential circuit and state-machine design by presenting several examples.

Early digital designers and many designers through the 1980s wrote out state tables by hand and built corresponding circuits using the synthesis methods that we described in Section 7.4. However, hardly anyone does that anymore. Nowadays, most state tables are specified with a hardware description language (HDL) such as ABEL, VHDL, or Verilog. The HDL compiler then performs the equivalent of our synthesis methods, and realizes the specified machine in a PLD, CPLD, FPGA, ASIC, or other target technology.

This chapter gives state-machine and other sequential-circuit design examples in two different HDLs. In the first section, we give examples in ABEL, and we target the resulting machines to small PLDs. Some of these examples illustrate the partitioning decisions that are needed when an entire circuit does not fit into a single component. In the second section, we give examples in VHDL, which can be targeted to just about any technology.

Like Chapter 6, this chapter has as prerequisites only the chapters that precede it. Its two sections are written to be pretty much independent of each other, and the rest of the book is written so that you can read this chapter now or skip it and come back later.

9.1 Design Examples Using ABEL and PLDs

In Section 7.4, we designed several simple state machines by translating their word descriptions into a state table, choosing a state assignment, and finally synthesizing a corresponding circuit. We repeated one of these examples using ABEL and a PLD in Table 7-27 on page 634 and another in Table 7-31 on page 637. These designs were much easier to do using ABEL and a PLD, for two reasons:

- You don’t have to be too concerned about the complexity of the resulting excitation equations, as long as they fit in the PLD.
- You may be able to take advantage of ABEL language features to make the design easier to express and understand.

Before looking at more examples, let’s examine the timing behavior and packaging considerations for state machines that are built from PLDs.

9.1.1 Timing and Packaging of PLD-Based State Machines

Figure 9-1 shows how a generic PLD with both combinational and registered outputs might be used in a state-machine application. Timing parameters t_{CO} and t_{PD} were explained in Section 8.3.3; t_{CO} is the flip-flop clock-to-output delay and t_{PD} is the delay through the AND-OR array.

State variables are assigned to registered outputs, of course, and are stable at time t_{CO} after the rising edge of CLOCK. Mealy-type outputs are assigned to combinational outputs, and are stable at time t_{PD} after any input change that

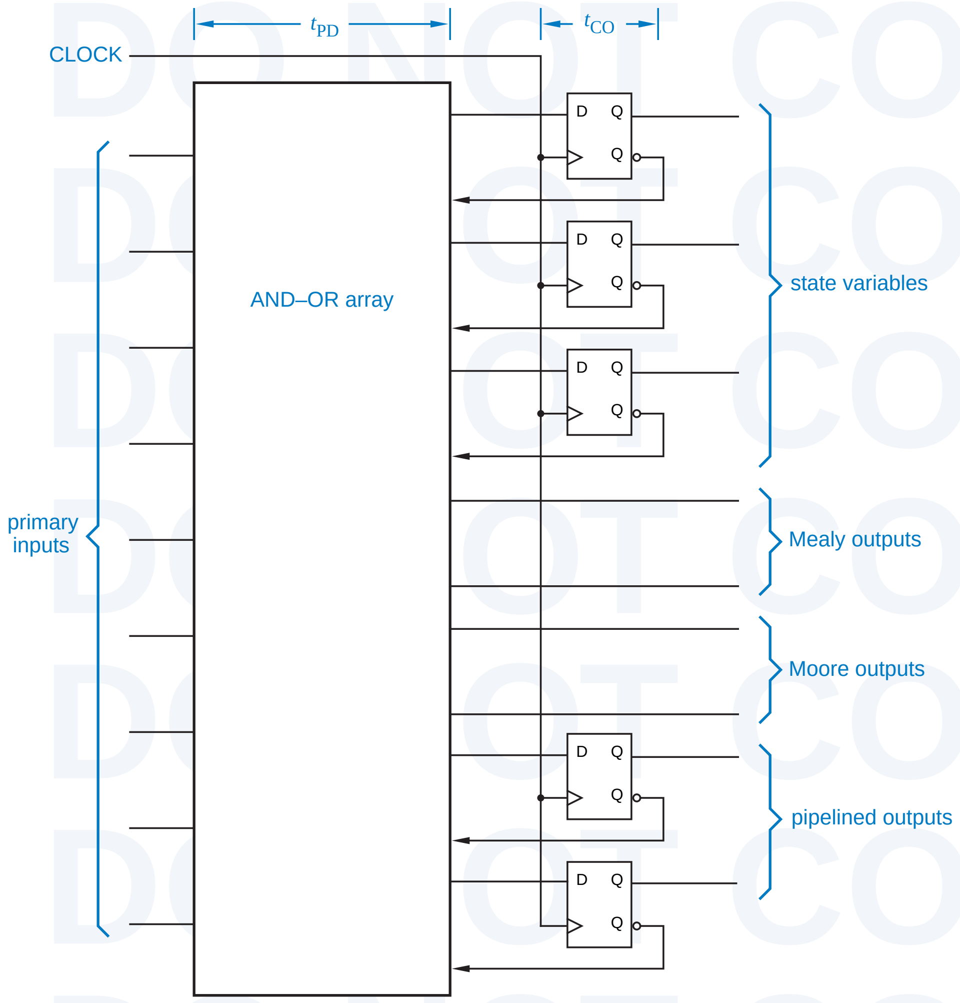


Figure 9-1
Structure and timing of a PLD used as a state machine.

affects them. Mealy-type outputs may also change after a state change, in which case they become stable at time $t_{CO} + t_{PD}$ after the rising edge of CLOCK.

A Moore-type output is, by definition, a combinational logic function of the current state, so it is also stable at time $t_{CO} + t_{PD}$ after the CLOCK. Thus, Moore outputs may not offer any speed advantage of Mealy outputs in a PLD realization. For faster propagation delay, we defined and used pipelined outputs in Sections 7.3.2 and 7.11.5. In a PLD realization, these outputs come directly from a flip-flop output, and thus have a delay of only t_{CO} from the clock. Besides having a shorter delay, they are also guaranteed to be glitch free, important in some applications.

PLD-based state-machine designs are often limited by the number of input and output pins available in a single PLD. According to the model of Figure 9-1,

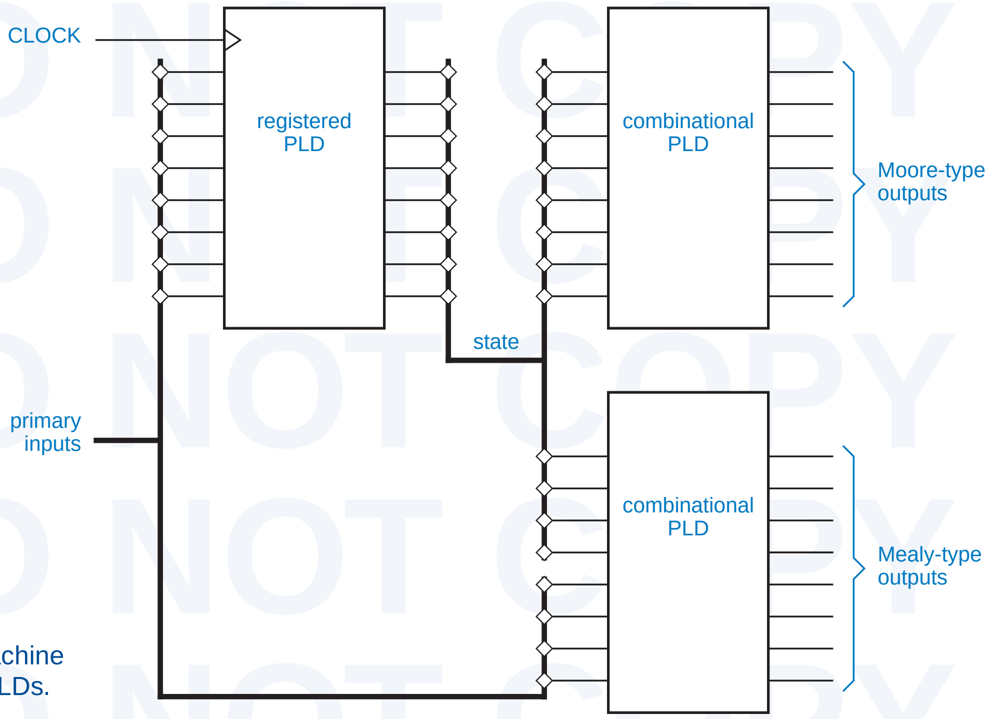


Figure 9-2
Splitting a state-machine design into three PLDs.

one PLD output is required for each state variable and for each Mealy- or Moore-type output. For example, the T-bird tail-lights machine of Section 7.5 starting on page 585 requires three registered outputs for the state variables and six combinational outputs for the lamp outputs, too many for most of the PLDs that we've described, except for the 22V10.

On the other hand, an output-coded state assignment (Section 7.3.2) usually requires a smaller total number of PLD outputs. Using an output-coded state assignment, the T-bird tail-lights machine can be built in a single 16V8, as we'll show in Section 9.1.3.

Like any state variable or Moore-type output, an output-coded state variable is stable at time t_{CO} after the clock edge. Thus, an output-coded state assignment improves speed as well as packaging efficiency. In T-bird tail lights, turning on the emergency flashers 10 ns sooner doesn't matter, but in a high-performance digital system, an extra 10 ns of delay could make the difference between a maximum system clock rate of 100 MHz and a maximum of only 50 MHz.

If a design *must* be split into two or more PLDs, the best split in terms of both design simplicity and packaging efficiency is often the one shown in Figure 9-2. A single sequential PLD is used to create the required next-state behavior, and combinational PLDs are used to create both Mealy- and Moore-type outputs from the state variables and inputs.

RELIEF FOR A SPLITTING HEADACHE

Modern software tools for PLD-based system design can eliminate some of the trial and error that might otherwise be associated with fitting a design into a set of PLDs. To use this capability, the designer enters the equations and state-machine descriptions for the design, without necessarily specifying the actual devices and pinouts that should be used to realize the design. A software tool called a *partitioner* attempts to split the design into the smallest possible number of devices from a given family, while minimizing the number of pins used for inter-device communication. Partitioning can be fully automatic, partially user controlled, or fully user controlled.

Larger devices—CPLDs and FPGAs—often have internal, architectural constraints that may create headaches in the absence of expert software assistance. It appear to the designer, based on input, output, and total combinational logic and flip-flip requirements, that a design will fit in a single CPLD or FPGA. However, the design must still be split among multiple PLDs or logic blocks inside the larger device, where each block has only limited functionality.

For example, an output that requires many product terms may have to steal some from physically adjacent outputs. This may in turn affect whether adjacent pins can be used as inputs or outputs, and how many product terms are available to them. It may also affect the ability to interconnect signals between nearby blocks and the worst-case delay of these signals. All of these constraints can be tracked by *fitter* software that uses a heuristic approach to find the best split of functions among the blocks within a single CPLD or FPGA.

In many design environments, the partitioner and fitter software work together and interactively with the designer to find an acceptable realization using a set of PLDs, CPLDs, and FPGAs.

9.1.2 A Few Simple Machines

In Section 7.4 we designed several simple state machines using traditional methods. We presented an ABEL- and PLD-based design for the first of these in Table 7-25 on page 631, and then we showed how we could make the machine's operation more understandable by making better use of ABEL features in Table 7-27 on page 634.

Our second example was a "1s-counting machine" with the following specification:

> Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z. The output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise.

We developed a state table for this machine in Table 7-12 on page 580. However, we can express the machine's function much more easily in ABEL, as shown in Table 9-1. Notice that for this machine we were better off not using ABEL's "state diagram" syntax. We could express the machine's function more naturally

Table 9-1
ABEL program for ones-counting state machine.

```
module onesctsm
title 'Ones-counting State Machine'
ONESCTSM device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, X, Y     pin 1, 2, 3, 4;
Z                      pin 13 istype 'com';
COUNT1..COUNT0         pin 14, 15 istype 'reg';

" Sets
COUNT = [COUNT1..COUNT0];

equations
COUNT.CLK = CLOCK;
WHEN RESET THEN COUNT := 0;
ELSE WHEN X & Y THEN COUNT := COUNT + 2;
ELSE WHEN X # Y THEN COUNT := COUNT + 1;
ELSE COUNT := COUNT;

Z = (COUNT == 0);

end onesctsm
```

using a nested WHEN statement and the built-in addition operation. The first WHEN clause forces the machine to its initial state and count of 0 upon reset, and the succeeding clauses increase the count by 2, 1, or 0 as required when the machine is not reset. Note that ABEL "throws away" the carry bit in addition, which is equivalent to performing addition modulo-4. The machine easily fits into a GAL16V8 device. It has four states, because there are two flip-flops its realization.

Another example from Section 7.4 is a combination-lock state machine (below we omit the HINT output in the original specification):

> Design a clocked synchronous state machine with one input, X, and one output, UNLK. The UNLK output should be 1 if and only if X is 0 and the sequence of inputs received on X at the preceding seven clock ticks was 0110111.

RESETTING BAD HABITS

In Chapter 7, we started the bad habit of designing state tables without including an explicit reset input. There was a reason for this—each additional input potentially doubles the amount work we would have had to do to synthesize the state machine using manual methods, and there was little to be gained pedagogically.

Now that we're doing language-based state-machine design using automated tools, we should get into the habit of always providing an explicit reset input that sends the machine back to a known state. It's a requirement in real designs!

We developed a state table for this machine in Table 7-14 on page 582, and we wrote an equivalent ABEL program in Table 7-31 on page 637. However, once again we can take a different approach that is easier to understand. For this example, we note that the output of the machine at any time is completely determined by its inputs over the preceding eight clock ticks. Thus, we can use a so-called "finite-memory" approach to design this machine, where we explicitly keep track of the past seven inputs and then form the output as a combinational function of these inputs.

The ABEL program in Table 9-2 uses the finite-memory approach. It is written using sets to make modifications easy, for example, to change the combination. However, note that the HINT output would be just as difficult to provide in this version of the machine as in the original (see Exercise 9.2).

Table 9-2 Finite-memory program for combination-lock state machine.

```
module comblckf
title 'Combination-Lock State Machine'
"COMBLCKF device ’P16V8R’;

" Input and output pins
CLOCK, RESET, X             pin 1, 2, 3;
X1..X7                      pin 12..18 istype 'reg';
UNLK                        pin 19;

" Sets
XHISTORY = [X7..X1];
SHIFTX  =  [X6..X1, X];

equations

XHISTORY.CLK = CLOCK;
XHISTORY := !RESET & SHIFTX;

UNLK = !RESET & (X == 0) & (XHISTORY == [0,1,1,0,1,1,1]);

END comblckf
```

FINITE-MEMORY DESIGN

The finite-memory design approach to state-machine design can be generalized. Such a machine's outputs are completely determined by its current input and outputs during the previous n clock ticks, where n is a finite, bounded integer. Any machine that can be realized as shown in Figure 9-3 is a finite-memory machine. Note that a finite-*state* machine need not be a finite-*memory* machine. For example, the ones-counting machine in Table 9-1 has only four states but is not a finite-memory machine, its output depends on every value of X and Y since reset.

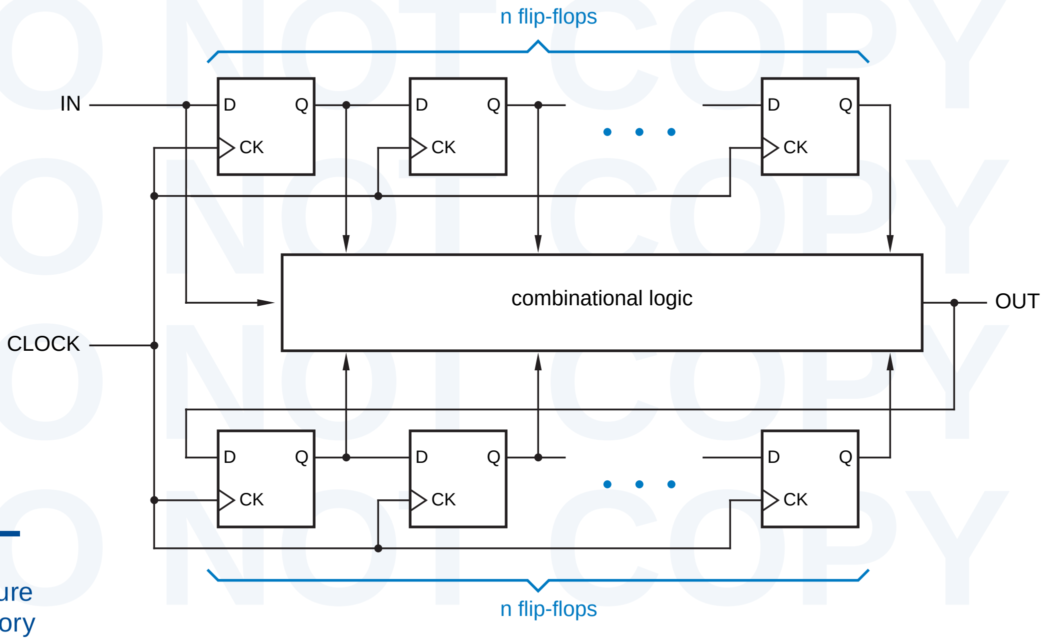


Figure 9-3 General structure of a finite-memory machine.

9.1.3 T-Bird Tail Lights

We described and designed a “T-bird tail-lights” state machine in Section 7.5. Table 9-3 is an equivalent ABEL “state diagram” for the T-bird tail-lights machine. There is a close correspondence between this program and the state diagram of Figure 7-64 on page 589 using the state assignment of Table 7-16 on page 590. Except for the added RESET input, the program produces exactly the same reduced equations as the explicit transition equations resulting from the transition list, which we worked out by hand in Section 7.6 on page 592.

The program in Table 9-3 handles only the state variables of the tail-lights machine. The output logic requires six combinational outputs, but only five more outputs are available in the 16V8 specified in the program. A second PLD could be used to decode the states, using the kind of partitioning that we showed in Figure 9-2. Alternatively, a larger PLD, such as the 22V10, could provide enough outputs for a single-PLD design.

An even better approach is to recognize that the output values of the tail-lights machine are different in each state, so we can also use an output-coded state assignment. This requires only six registered outputs and no combinational outputs of a 16V8, as shown in Figure 9-4. Only the device, pin, and state definitions in the previous ABEL program must be changed, as shown in Table 9-4. The six resulting excitation equations each use four product terms.

```
module tbirdsd
title 'State Machine for T-Bird Tail Lights'
TBIRDSD device 'P16V8R';

" Input and output pins
CLOCK, LEFT, RIGHT, HAZ, RESET   pin 1, 2, 3, 4, 5;
Q0, Q1, Q2                       pin 14, 15, 16 istype 'reg';

" Definitions
QSTATE = [Q2,Q1,Q0];            " State variables
IDLE   = [ 0, 0, 0];            " States
L1     = [ 0, 0, 1];
L2     = [ 0, 1, 1];
L3     = [ 0, 1, 0];
R1     = [ 1, 0, 1];
R2     = [ 1, 1, 1];
R3     = [ 1, 1, 0];
LR3    = [ 1, 0, 0];

equations
QSTATE.CLK = CLOCK;

state_diagram QSTATE
state IDLE: IF RESET THEN IDLE
            ELSE IF (HAZ # LEFT & RIGHT) THEN LR3
            ELSE IF LEFT THEN L1 ELSE IF RIGHT THEN R1
            ELSE IDLE;
state L1:   IF RESET THEN IDLE ELSE IF HAZ THEN LR3 ELSE L2;
state L2:   IF RESET THEN IDLE ELSE IF HAZ THEN LR3 ELSE L3;
state L3:   GOTO IDLE;
state R1:   IF RESET THEN IDLE ELSE IF HAZ THEN LR3 ELSE R2;
state R2:   IF RESET THEN IDLE ELSE IF HAZ THEN LR3 ELSE R3;
state R3:   GOTO IDLE;
state LR3:  GOTO IDLE;

end tbirdsd
```

Table 9-3 ABEL program for the T-bird tail-lights machine.

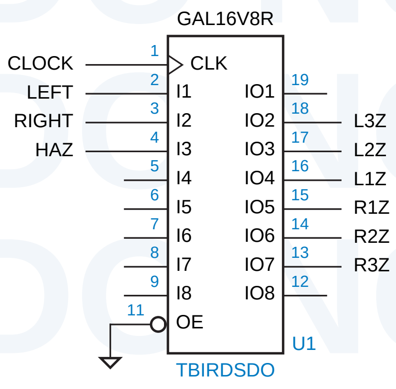


Figure 9-4 A single-PLD design for T-bird tail lights.

Table 9-4
Output-coded state assignment for the T-bird tail-lights machine.

```
module tbirdsdo
title 'Output-Coded T-Bird Tail Lights State Machine'
TBIRDSDO device 'P16V8R';

" Input and output pins
CLOCK, LEFT, RIGHT, HAZ, RESET     pin 1, 2, 3, 4, 5;
L3Z, L2Z, L1Z, R1Z, R2Z, R3Z       pin 18..13 istype 'reg';

" Definitions
QSTATE = [L3Z,L2Z,L1Z,R1Z,R2Z,R3Z];  " State variables
IDLE   = [  0,  0,  0,  0,  0,  0];  " States
L3     = [  1,  1,  1,  0,  0,  0];
L2     = [  0,  1,  1,  0,  0,  0];
L1     = [  0,  0,  1,  0,  0,  0];
R1     = [  0,  0,  0,  1,  0,  0];
R2     = [  0,  0,  0,  1,  1,  0];
R3     = [  0,  0,  0,  1,  1,  1];
LR3    = [  1,  1,  1,  1,  1,  1];
```

9.1.4 The Guessing Game

A “guessing game” machine was defined in Section 7.7.1 starting on page 594, with the following description:

> Design a clocked synchronous state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to lamps or LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red lamp. In normal operation, the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.
>
> Guesses are made by pressing a pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the “wrong” pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the lamp output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

As we discussed in Section 7.7.1, the machine requires six states—four in which a corresponding lamp is on, and two for when play is stopped after either a good or a bad pushbutton push. An ABEL program for the guessing game is shown in Table 9-5. Two enhancements were made to improve the testability and robustness of the machine—a RESET input that forces the game to a known starting state, and the two unused states have explicit transitions to the starting state.

The guessing-game machine uses the same state assignments as the original version in Section 7.7.1. Using these assignments, the ABEL compiler

Table 9-5
ABEL program for the guessing-game machine.

```
module ggame
Title 'Guessing-Game State Machine'
GGAME device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, G1..G4          pin 1, 2, 3..6;
L1..L4, ERR                   pin 12..15, 19 istype 'com';
Q2..Q0                        pin 16..18 istype 'reg';

" Sets
G = [G1..G4];
L = [L1..L4];

" States
QSTATE = [Q2,Q1,Q0];
S1     = [ 0, 0, 0];
S2     = [ 0, 0, 1];
S3     = [ 0, 1, 1];
S4     = [ 0, 1, 0];
SOK    = [ 1, 0, 0];
SERR   = [ 1, 0, 1];
EXTRA1 = [ 1, 1, 0];
EXTRA2 = [ 1, 1, 1];

state_diagram QSTATE

state S1:  IF RESET THEN SOK ELSE IF G2 # G3 # G4 THEN SERR
           ELSE IF G1 THEN SOK ELSE S2;

state S2:  IF RESET THEN SOK ELSE IF G1 # G3 # G4 THEN SERR
           ELSE IF G2 THEN SOK ELSE S3;

state S3:  IF RESET THEN SOK ELSE IF G1 # G2 # G4 THEN SERR
           ELSE IF G3 THEN SOK ELSE S4;

state S4:  IF RESET THEN SOK ELSE IF G1 # G2 # G3 THEN SERR
           ELSE IF G4 THEN SOK ELSE S1;

state SOK:  IF RESET THEN SOK
            ELSE IF G1 # G2 # G3 # G4 THEN SOK ELSE S1;

state SERR:  IF RESET THEN SOK
             ELSE IF G1 # G2 # G3 # G4 THEN SERR ELSE S1;

state EXTRA1:  GOTO SOK;
state EXTRA2:  GOTO SOK;

equations

QSTATE.CLK = CLOCK;

L1  = (QSTATE == S1);
L2  = (QSTATE == S2);
L3  = (QSTATE == S3);
L4  = (QSTATE == S4);
ERR = (QSTATE == SERR);

end ggame
```

Table 9-6
Product-term usage in the guessing-game state-machine PLD.

```
 P-Terms     Fan-in  Fan-out  Type  Name
---------    ------  -------  ----  --------
   1/3         3        1     Pin   L1
   1/3         3        1     Pin   L2
   1/3         3        1     Pin   L3
   1/3         3        1     Pin   L4
   1/3         3        1     Pin   ERR
   6/2         7        1     Pin   Q2.REG
   1/7         7        1     Pin   Q1.REG
  11/8         8        1     Pin   Q0.REG
=========
  23/32          Best P-Term Total: 16
                        Total Pins: 14
              Average P-Term/Output: 2
```

cranks out minimized equations with the number of product terms shown in Table 9-6. The Q0 output just barely fits in a GAL16V8 (eight product terms). If we needed to save terms, the way in which we've written the program allows us to try alternate state assignments (see Exercise 9.4).

A more productive alternative might be to try an output-coded state assignment. We can use one state/output bit per lamp (L1..L4), and use one more bit (ERR) to distinguish between the SOK and SERR states when the lamps are all off. This allows us to drop the equations for L1..L4 and ERR from Table 9-5. The new assignment is shown in Table 9-7. With this assignment, L1 uses two product terms and L2..L4 use only one product term each. Unfortunately, the ERR output blows up into 16 product terms.

Table 9-7
ABEL definitions for the guessing-game machine with an output-coded state assignment.

```
module ggameoc
Title 'Guessing-Game State Machine'
"GGAMEOC device 'P16V8R';

" Inputs and outputs
CLOCK, RESET, G1..G4             pin 1, 2, 3..6;
L1..L4, ERR                      pin 12..15, 18 istype 'reg';

" States
QSTATE = [L1,L2,L3,L4,ERR];
S1     = [ 1, 0, 0, 0,  0];
S2     = [ 0, 1, 0, 0,  0];
S3     = [ 0, 0, 1, 0,  0];
S4     = [ 0, 0, 0, 1,  0];
SOK    = [ 0, 0, 0, 0,  0];
SERR   = [ 0, 0, 0, 0,  1];

...
```

Part of our problem with this particular output-coded assignment is that we’re not taking full advantage of its properties. Notice that it is basically a “one-hot” encoding, but the state definitions in Table 9-7 require all five state bits to be decoded for each state. An alternate version of the coding using “don’t-cares” is shown in Table 9-8.

In the new version, we are assuming that the state bits never take on any combination of values other than the ones we originally defined in Table 9-7. Thus, for example, if we see that state bit L1 is 1, the machine must be in state S1 regardless of the values of any other state bits. Therefore, we can set these bits to “don’t care” in S1’s definition in Table 9-8. ABEL will set each X to 0 when encoding a next state, but will treat each X as a “don’t-care” when decoding the current state. Thus, we must take *extreme* care to ensure that decoded states are in fact mutually exclusive, that is, that no legitimate next state matches two or more different state definitions. Otherwise, the compiled results will not have the expected behavior.

The reduced equations that result from the output coding in Table 9-8 use three product terms for L1, one each for L2..L4, and only seven for ERR. So the

```
X = .X.;
QSTATE = [L1,L2,L3,L4,ERR];
S1     = [ 1, X, X, X,  X];
S2     = [ X, 1, X, X,  X];
S3     = [ X, X, 1, X,  X];
S4     = [ X, X, X, 1,  X];
SOK    = [ 0, 0, 0, 0,  0];
SERR   = [ X, X, X, X,  1];
```

Table 9-8 Output coding for the guessing-game machine using “don’t cares.”

DON’T-CARE, HOW IT WORKS

To understand how the don’t-cares work in a state encoding, you must first understand how ABEL creates equations internally from state diagrams. Within a given state S, each transition statement (IF-THEN-ELSE or GOTO) causes the on-sets of certain state variables to be augmented according to the transition condition. The transition condition is an expression that must be true to “go to” that target, including being in state S. For example, all of the conditions specified in a state such as S1 in Table 9-5 are implicitly ANDed with the expression “QSTATE==S1”. Because of the way S1 is defined using don’t-cares, this equality check generates only a single literal (L1) instead of an AND term, leading to further simplification later.

For each target state in a transition statement, the on-sets of only the state variables that are 1 in that state are augmented according to the transition condition. Thus, when a coded state such as S1 in Table 9-8 appears as a target in any transition statement, only the on-set of L1 is augmented. This explains why the actual coding of state S1 as a target is 100000.

change was worthwhile. However, we must remember that the new machine is different from the one in Table 9-7. Consider what happens if the machine ever gets into an unspecified state. In the original machine with fully specified output coding, there are no next-states for the $2^5 - 6 = 26$ unspecified states, so the state machine will always go to the state coded 00000 (SOK) from unspecified states. In the new machine, “unspecified” states aren’t really unspecified; for example, the state coded 11111 actually matches five coded states, S1–S4 and SERR. The next state will actually be the “OR” of next-states for the matching coded states. (Read the box on the previous page to understand why these outcomes occur.) Again, you need to be careful.

RESETTING EXPECTATIONS

Reading the guessing-game program in Table 9-5, you would expect that the RESET input would force the machine to the SOK state, and it does. However, the moment that you have unspecified or partially coded states as in Tables 9-7 or 9-8, don’t take anything for granted.

Referring to the box on the previous page, remember that transition statements in ABEL state machines augment the on-sets of state variables. If a particular, unused state combination does not match any of the states for which transition statements were written, then no on-sets will be augmented. Thus, the only transition from that state will be to the state with the all-0s coding.

For this reason, it is useful to code the reset state or a “safe” state as all 0s. If this is not possible, but the all-0s state is still unused, you can explicitly provide a transition from the all-0s state to a desired safe state.

9.1.5 Reinventing Traffic-Light Controllers

Our final example is from the world of cars and traffic. Traffic-light controllers in California, especially in the fair city of Sunnyvale, are carefully designed to *maximize* the waiting time of cars at intersections. An infrequently used intersection (one that would have no more than a “yield” sign if it were in Chicago) has the sensors and signals shown in Figure 9-5. The state machine that controls the traffic signals uses a 1 Hz clock and a timer and has four inputs:

NSCAR	Asserted when a car on the north-south road is over either sensor on either side of the intersection.
EWCAR	Asserted when a car on the east-west road is over either sensor on either side of the intersection.
TMLONG	Asserted if more than five minutes has elapsed since the timer started; remains asserted until the timer is reset.
TMSHORT	Asserted if more than five seconds has elapsed since the timer started; remains asserted until the timer is reset.

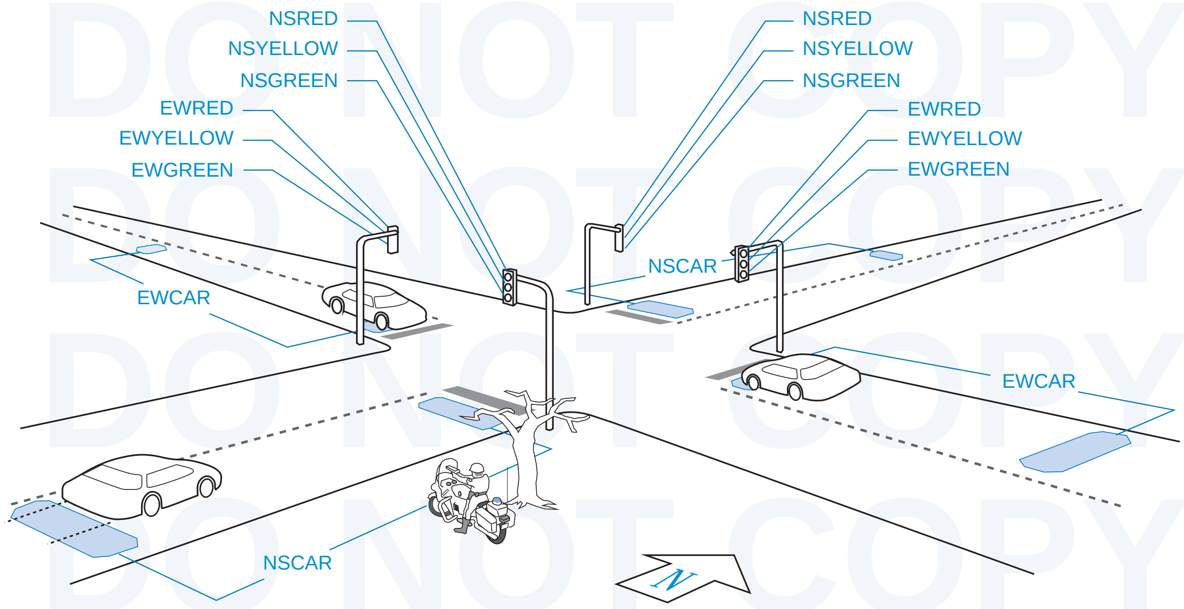


Figure 9-5 Traffic sensors and signals at an intersection in Sunnyvale, California.

The state machine has seven outputs:

NSRED, NSYELLOW, NSGREEN	Control the north-south lights.
EWRED, EWYELLOW, EWGREEN	Control the east-west lights.
TMRESET	When asserted, resets the timer and negates TMSHORT and TMLONG. The timer starts timing when TMRESET is negated.

A typical, municipally approved algorithm for controlling the traffic lights is embedded in the ABEL program of Table 9-9. This algorithm produces two frequently seen behaviors of “smart” traffic lights. At night, when traffic is light, it holds a car stopped at the light for up to five minutes, unless a car approaches on the cross street, in which case it stops the cross traffic and lets the waiting car go. (The “early warning” sensor is far enough back to change the lights before the approaching car reaches the intersection.) During the day, when traffic is heavy and there are always cars waiting in both directions, it cycles the lights every five seconds, thus minimizing the utilization of the intersection and maximizing everyone’s waiting time, thereby creating a public outcry for more taxes to fix the problem.

The equations for the TMRESET output are worth noting. This output is asserted during the “double-red” states, NSDELAY and EWDELAY, to reset the timer in preparation for the next green cycle. The desired output signal could be generated on a combinational output pin by decoding these two states, but we have chosen instead to generate it on a registered output pin by decoding the *predecessors* of these two states.

Table 9-9 Sunnyvale traffic-lights program.

```
module svaletl
title 'State Machine for Sunnyvale, CA, Traffic Lights'
SVALETL device ’P16V8R’;

" Input and output pins
CLOCK, !OE                        pin 1, 11;
NSCAR, EWCAR, TMSHORT, TMLONG     pin 2, 3, 8, 9;
Q0, Q1, Q2, TMRESET_L             pin 17, 16, 15, 14 istype 'reg';

" Definitions
LSTATE  = [Q2,Q1,Q0];            " State variables
NSGO    = [ 0, 0, 0];            " States
NSWAIT  = [ 0, 0, 1];
NSWAIT2 = [ 0, 1, 1];
NSDELAY = [ 0, 1, 0];
EWGO    = [ 1, 1, 0];
EWWAIT  = [ 1, 1, 1];
EWWAIT2 = [ 1, 0, 1];
EWDELAY = [ 1, 0, 0];

state_diagram LSTATE
state NSGO:                      " North-south green
   IF (!TMSHORT) THEN NSGO       " Minimum green is 5 seconds.
   ELSE IF (TMLONG) THEN NSWAIT  " Maximum green is 5 minutes.
   ELSE IF (EWCAR & !NSCAR)      " If E-W car is waiting and no one
          THEN NSGO              "  is coming N-S, make E-W wait!
   ELSE IF (EWCAR & NSCAR)       " Cars coming in both directions?
          THEN NSWAIT            " Thrash!
   ELSE IF (!NSCAR)              " Nobody coming N-S and not timed out?
          THEN NSGO              " Keep N-S green.
   ELSE NSWAIT;                  " Else let E-W have it.

state NSWAIT: GOTO NSWAIT2;      " Yellow light is on for two ticks for safety.
state NSWAIT2: GOTO NSDELAY;     " (Drivers go 70 mph to catch this turkey green!)
state NSDELAY: GOTO EWGO;        " Red in both directions for added safety.

state EWGO: " East-west green; states defined analogous to N-S
   IF (!TMSHORT) THEN EWGO
   ELSE IF (TMLONG) THEN EWWAIT
   ELSE IF (NSCAR & !EWCAR) THEN EWGO
   ELSE IF (NSCAR & EWCAR) THEN EWWAIT
   ELSE IF (!EWCAR) THEN EWGO  ELSE EWWAIT;

state EWWAIT: GOTO EWWAIT2;
state EWWAIT2: GOTO EWDELAY;
state EWDELAY: GOTO NSGO;

equations

LSTATE.CLK = CLOCK;  TMRESET_L.CLK = CLOCK;
!TMRESET_L := (LSTATE == NSWAIT2)  " Reset the timer when going into
            + (LSTATE == EWWAIT2); "   state NSDELAY or state EWDELAY.
end svaletl
```

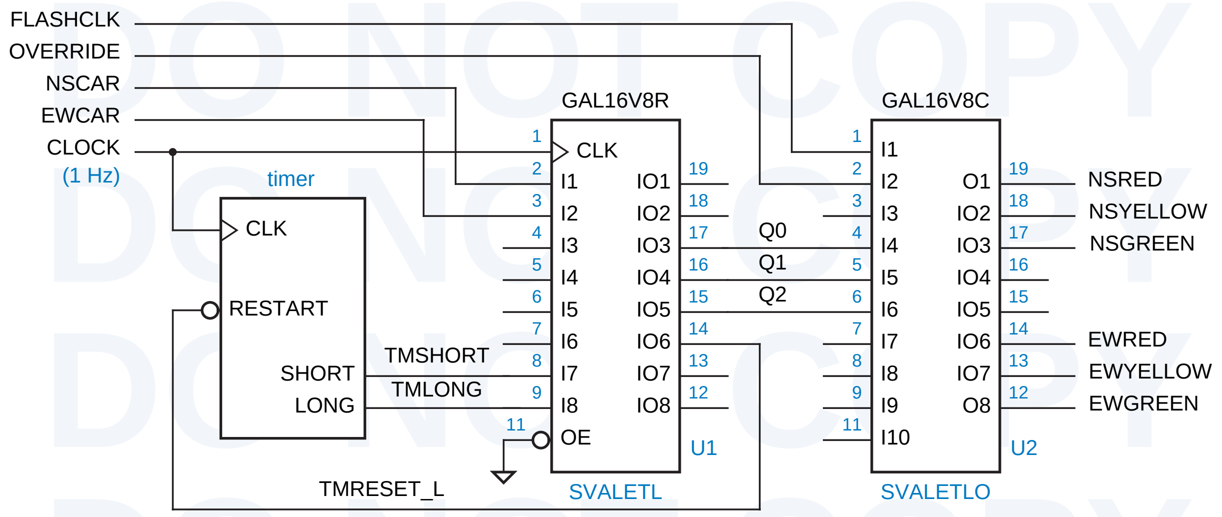


Figure 9-6 Sunnyvale traffic-light controller using two PLDs.

Table 9-10 Output logic for Sunnyvale traffic lights.

```
module svaletlo
title 'Output logic for Sunnyvale, CA, Traffic Lights'
"SVALETLO device ’P16V8C’;

" Input pins
FLASHCLK, OVERRIDE, Q0, Q1, Q2   pin 1, 2, 4, 5, 6;

" Output pins
NSRED, NSYELLOW, NSGREEN         pin 19, 18, 17 istype 'com';
EWRED, EWYELLOW, EWGREEN         pin 14, 13, 12 istype 'com';

" Definitions (same as in state machine SVALETL)
...

equations

NSRED = !OVERRIDE & (LSTATE != NSGO) & (LSTATE != NSWAIT) & (LSTATE != NSWAIT2)
      # OVERRIDE & FLASHCLK;
NSYELLOW = !OVERRIDE & ((LSTATE == NSWAIT) # (LSTATE == NSWAIT2));
NSGREEN  = !OVERRIDE & (LSTATE == NSGO);

EWRED = !OVERRIDE & (LSTATE != EWGO) & (LSTATE != EWWAIT) & (LSTATE != EWWAIT2)
      # OVERRIDE & FLASHCLK;
EWYELLOW = !OVERRIDE & ((LSTATE == EWWAIT) # (LSTATE == EWWAIT2));
EWGREEN  = !OVERRIDE & (LSTATE == EWGO);

end svaletlo
```

The ABEL program in Table 9-9 defines only the state variables and one registered Moore output for the traffic controller. Six more Moore outputs are needed for the lights, more than remain on the 16V8. Therefore, a separate combinational PLD is used for these outputs, yielding the complete design shown in Figure 9-6 on the preceding page. An ABEL program for the output PLD is given in Table 9-10. We've taken this opportunity to add an OVERRIDE input to the controller. This input may be asserted by the police to disable the controller and put the signals into a flashing-red mode (at a rate determined by FLASHCLK), allowing them to manually clear up the traffic snarls created by this wonderful invention.

A traffic-light state machine including output logic can be built in a single 16V8, shown in Figure 9-7, if we choose an output-coded state assignment. Only the definitions in the original program of Table 9-9 must be changed, as shown in Table 9-11. This PLD does not include the OVERRIDE input and mode, which is left as an exercise (9.7).

Table 9-11 Definitions for Sunnyvale traffic-lights machine with output-coded state assignment.

```
module svaletlb
title 'Output-Coded State Machine for Sunnyvale Traffic Lights'
"SVALETLB device 'P16V8R';

" Input and output pins
CLOCK, !OE                          pin 1, 11;
NSCAR, EWCAR, TMSHORT, TMLONG       pin 2, 3, 8, 9;
NSRED, NSYELLOW, NSGREEN            pin 19, 18, 17 istype 'reg';
EWRED, EWYELLOW, EWGREEN            pin 16, 15, 14 istype 'reg';
TMRESET_L, XTRA                     pin 13, 12 istype 'reg';

" Definitions
LSTATE   = [NSRED,NSYELLOW,NSGREEN,EWRED,EWYELLOW,EWGREEN,XTRA]; " State vars
NSGO     = [    0,       0,      1,    1,      0,      0,   0]; " States
NSWAIT   = [    0,       1,      0,    1,      0,      0,   0];
NSWAIT2  = [    0,       1,      0,    1,      0,      0,   1];
NSDELAY  = [    1,       0,      0,    1,      0,      0,   0];
EWGO     = [    1,       0,      0,    0,      0,      1,   0];
EWWAIT   = [    1,       0,      0,    0,      1,      0,   0];
EWWAIT2  = [    1,       0,      0,    0,      1,      0,   1];
EWDELAY  = [    1,       0,      0,    1,      0,      0,   1];
```

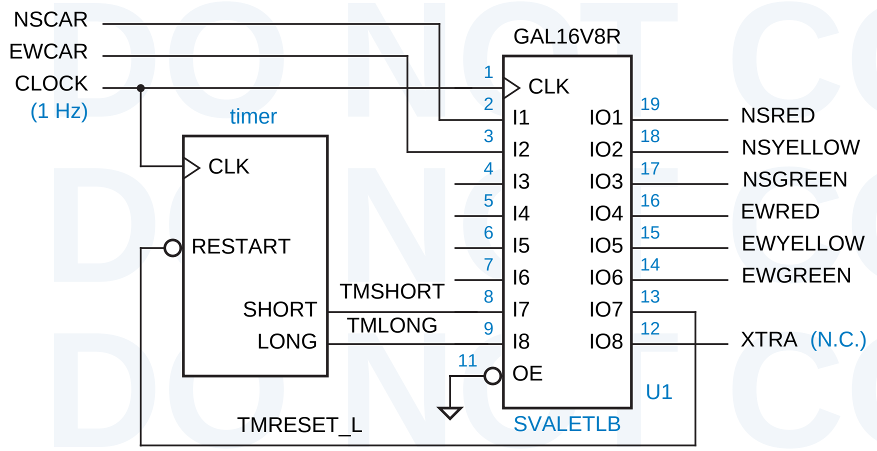


Figure 9-7 Traffic-light state machine using output-coded state assignment in a single PLD.

9.2 Design Examples Using VHDL

As we explained Section 7.12, the basic VHDL language features that we introduced way back in Section 4.7, including processes, are just about all that is needed to model sequential-circuit behavior. Unlike ABEL, VHDL does not provide any special language elements for modeling state machines. Instead, most programmers use a combination of existing “standard” features—most notably enumerated types and case statements—to write state-machine descriptions. We’ll use this method in the examples in this section.

9.2.1 A Few Simple Machines

In Section 7.4.1 we illustrated the state-table design process using the simple design problem below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> – A had the same value at each of the two previous clock ticks, *or*
>
> – B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

In an HDL-based design environment, there are many possible ways of writing a program that meets the stated requirements. We’ll look at several.

The first approach is to construct a state and output table by hand, and then manually convert it into a corresponding program. Since we already developed such a state table in Section 7.4.1, why not use it? We’ve written it again in Table 9-12, and we’ve written a corresponding VHDL program in Table 9-13.

Table 9-12
State and output table for the example state machine.

	A B				
S	00	01	11	10	Z
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S*				

Table 9-13
VHDL program for state-machine example.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity smexamp is
  port ( CLOCK, A, B: in STD_LOGIC;
         Z: out STD_LOGIC );
end;

architecture smexamp_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
signal Sreg: Sreg_type;
begin

  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = '1' then
      case Sreg is
        when INIT => if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= A1;   end if;
        when A0 =>   if    A='0' then Sreg <= OK0;
                     elsif A='1' then Sreg <= A1;  end if;
        when A1 =>   if    A='0' then Sreg <= A0;
                     elsif A='1' then Sreg <= OK1;  end if;
        when OK0 =>  if    A='0' then Sreg <= OK0;
                     elsif A='1' and B='0' then Sreg <= A1;
                    elsif A='1' and B='1' then Sreg <= OK1;  end if;
        when OK1 =>  if    A='0' and B='0' then Sreg <= A0;
                     elsif A='0' and B='1' then Sreg <= OK0;
                    elsif A='1' then Sreg <= OK1;            end if;
        when others => Sreg <= INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <= '0' when INIT | A0 | A1,
         '1' when OK0 | OK1,
         '0' when others;

end smexamp_arch;
```

As usual, the VHDL entity declaration specifies only inputs and outputs—CLOCK, A, B, and Z in this example. The architecture definition specifies the state machine's internal operation. The first thing it does is to create an enumerated type, Sreg_type, whose values are identifiers corresponding to the state names. Then it declares a signal, Sreg, which will be used to hold the machine's current state. Because of the way Sreg is used later, it will map into an edge-triggered register in synthesis.

The statement part of the architecture has two concurrent statements—a process and a selected-assignment statement. The process is sensitive only to CLOCK and establishes all of the state transitions, which occur on the rising edge of CLOCK. Within the process, an "if" statement checks for the rising edge, and a case statement enumerates the transitions for each state.

The case statement has six cases, corresponding to the five explicitly defined states and a catch-all for other states. For robustness, the "others" case sends the machine back to the INIT state. In each case we've used a nested "if" statement to explicit cover all combinations of inputs A and B. However, it's not strictly necessary to include the combinations where there is no transition from the current state; Sreg will hold its current value if no assignment is made on a particular process invocation.

The selected-assignment statement at the end of Table 9-13 handles the machine's single Moore output, Z, which is set to a value as a function of the current state. It would be easy to define Mealy outputs within this statement as well. That is, Z could be a function of the inputs as well as the current state. Since the "with" statement is a concurrent state, any input changes will affect the Z output as soon as they occur.

We really should have included a reset input in Table 9-13 (see box on page 808). A RESET signal is easily accommodated by modifying the entity declaration, and adding a clause to the "if" statement in the architecture definition. If RESET is asserted, the machine should go to the INIT state, otherwise the case statement should be executed. Depending on whether we check RESET before or after the clock-edge check, we can create either an asynchronous or a synchronous reset behavior (see Exercise 9.10).

What about the state assignment problem? Table 9-13 gives no information on how state-variable combinations are to be assigned to the named states, or even how many binary state variables are needed in the first place!

A synthesis tool is free to associate any integer values or binary combinations it likes with the identifiers in an enumerated type, but a typical tool will assign integers in the order the state names are listed, starting with 0. It will then use the smallest possible number of bits to encode those integers, that is, $\lceil \log_2 s \rceil$ bits for s states. Thus, the program in Table 9-13 will typically synthesize with the same, "simplest" state assignment that we chose in the original example in Table 7-7 on page 571. However, VHDL supports a couple of ways that we can force a different assignment.

Table 9-14
Using an attribute to force an enumeration encoding.

```
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;
...
architecture smexampe_arch of smexamp is
type Sreg_type is (INIT, A0, A1, OK0, OK1);
attribute enum_encoding of Sreg_type: type is
          "0000 0001 0010 0100 1000";
signal Sreg: Sreg_type;
...
```

One way to force an assignment is to use VHDL's "attribute" statement as shown in Table 9-14. Here, "enum_encoding" is a user-defined attribute whose value is a string that specifies the enumeration encoding to be used by the synthesis tool. The VHDL language processor ignores this value, but passes the attribute name and its value to the synthesis tool. The attribute "enum_encoding" is defined and known by most synthesis tools, including tools from Synopsys, Inc. Notice that a Synopsys "attributes" package must be "used" by the program; this is necessary for the VHDL compiler to recognize "enum_encoding" as a legitimate user-defined attribute. By the way, the state assignment that we've specified in the program is equivalent to the "almost one-hot" coding in Table 7-7 on page 571.

Another way to force an assignment, without relying on external packages or synthesis attributes, is to define the state register more explicitly using standard logic data types. This approach is shown in Table 9-15. Here, Sreg is defined as a 4-bit STD_LOGIC_VECTOR, and constants are defined to allow the states to referenced by name elsewhere in the program. No other changes in the program are required.

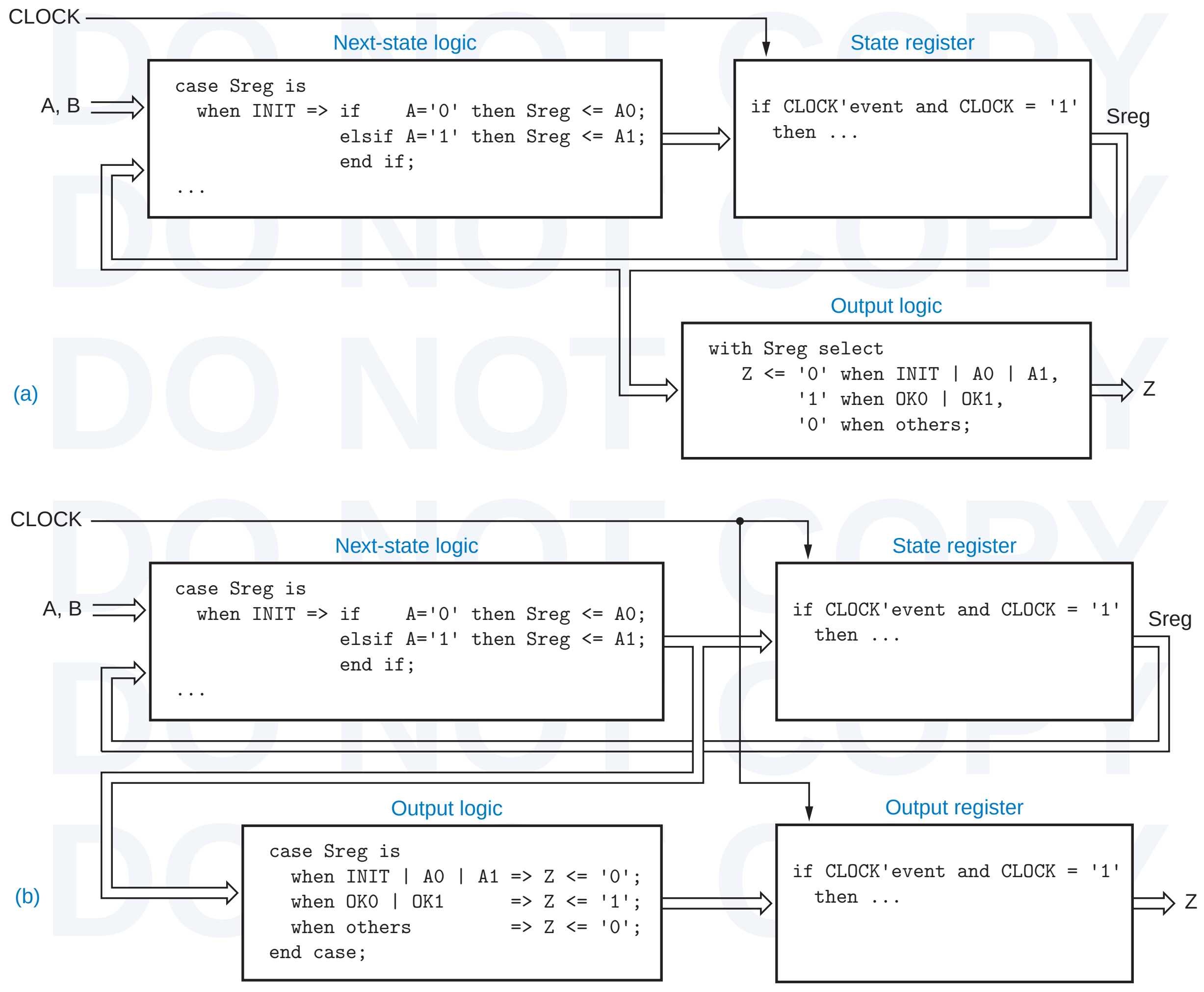
Going back to our original VHDL program in Table 9-13, one more interesting change is possible. As written, the program defines a conventional Moore-type state machine with the structure shown in Figure 9-8(a). What

Table 9-15
Using standard logic and constants to specify a state encoding.

```
library IEEE;
use IEEE.std_logic_1164.all;
...
architecture smexampc_arch of smexamp is
subtype Sreg_type is STD_LOGIC_VECTOR (1 to 4);
constant INIT: Sreg_type := "0000";
constant A0  : Sreg_type := "0001";
constant A1  : Sreg_type := "0010";
constant OK0 : Sreg_type := "0100";
constant OK1 : Sreg_type := "1000";
signal Sreg: Sreg_type;
...
```

happens if, we convert the output logic's selected-assignment statement into a `case` statement and move it into the state-transition process? By doing this, we create a machine that will most likely be synthesized with the structure shown in (b). This is essentially a Mealy machine with pipelined outputs whose behavior is indistinguishable from that of the original machine, except for timing. We've reduced the propagation delay from CLOCK to Z by producing Z directly on a register output, but we've also increased the setup-time requirements of A and B to CLOCK, because of the extra propagation delay through the output logic to the D input of the output register.

Figure 9-8 State-machine structures implied by VHDL programs: (a) Moore machine with combinational output logic; (b) pipelined Mealy machine with output register.



(a) Next-state logic:

```
case Sreg is
  when INIT => if    A='0' then Sreg <= A0;
               elsif A='1' then Sreg <= A1;
               end if;
...
```

(a) State register:

```
if CLOCK'event and CLOCK = '1'
  then ...
```

(a) Output logic:

```
with Sreg select
   Z <= '0' when INIT | A0 | A1,
        '1' when OK0 | OK1,
        '0' when others;
```

(b) Next-state logic:

```
case Sreg is
  when INIT => if    A='0' then Sreg <= A0;
               elsif A='1' then Sreg <= A1;
               end if;
...
```

(b) State register:

```
if CLOCK'event and CLOCK = '1'
  then ...
```

(b) Output logic:

```
case Sreg is
  when INIT | A0 | A1 => Z <= '0';
  when OK0 | OK1      => Z <= '1';
  when others         => Z <= '0';
end case;
```

(b) Output register:

```
if CLOCK'event and CLOCK = '1'
  then ...
```

TRICKY TIMING When we write a VHDL architecture corresponding to Figure 9-8(b), it is very important to add `Sreg` to the sensitivity list of the process. Notice that the output-logic `case` statement determines the value of `Z` as a function of `Sreg`. Throughout the first execution of the process after a rising clock edge, `Sreg` is contains the old state value. This is true because `Sreg` is a signal, not a variable. As explained in Section 4.7.9, signals that are changed in a process do not acquire their new values until at least one delta delay *after* the process has executed. By putting `Sreg` in the sensitivity list, we ensure that the process is executed a second time, so the final value of `Z` is based on the new value of `Sreg`.

All of the solutions to the example state-machine design problem that we've shown so far rely on the state table that we originally constructed by hand in Section 7.4.1. However, it is possible to write a VHDL program directly, without writing out a state table by hand.

Based on the original problem statement on page 813, the key simplifying idea is to remove the last value of `A` from the state definitions, and instead to have a separate register that keeps track of it (`LASTA`). Then only two non-`INIT` states

Table 9-16 Simplified state machine for VHDL example problem.

```
architecture smexampa_arch of smexamp is
type Sreg_type is (INIT, LOOKING, OK);
signal Sreg: Sreg_type;
signal lastA: STD_LOGIC;
begin

  process (CLOCK) -- state-machine states and transitions
  begin
    if CLOCK'event and CLOCK = '1' then
      lastA <= A;
      case Sreg is
        when INIT    =>                      Sreg <= LOOKING;
        when LOOKING => if    A=lastA then Sreg <= OK;
                        else               Sreg <= LOOKING;
                        end if;
        when OK      => if    B='1'   then Sreg <= OK;
                        elsif A=lastA then Sreg <= OK;
                        else               Sreg <= LOOKING;
                        end if;
        when others  =>                    Sreg <= INIT;
      end case;
    end if;
  end process;

  with Sreg select  -- output values based on state
    Z <= '1' when OK,
         '0' when others;

end smexampa_arch;
```

must be defined: LOOKING (“still looking for a match”) and OK (“got a match or B has been 1 since last match”). A VHDL architecture based on this approach is shown in Table 9-16. In the CLOCK-driven process, the first assignment statement creates the LASTA register, and the case statement creates the 3-state machine. At the end of the program, the Z output is defined as a simple combinational decode of the OK state.

Another simple state-machine example is a “1s-counting machine” with the following specification:

> Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z. The output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise.

We developed a state table for this machine in Table 7-12 on page 580. However, we can make use of the counting capabilities in the IEEE std_logic_arith package to write a VHDL program for this problem directly.

Table 9-17 shows our solution. As always, there are many different ways to solve the problem, and we have picked a way that illustrates several different language features. Within the architecture, we declare a subtype COUNTER which is a 2-bit UNSIGNED value. We then declare a signal COUNT of this type to hold the ones count, and a constant ZERO of the same type for initializing and checking the value of COUNT.

Table 9-17 VHDL program for a ones-counting machine.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity Vonescnt is
  port ( CLOCK, RESET, X, Y: in STD_LOGIC;
         Z: out STD_LOGIC );
end;

architecture Vonescnt_arch of Vonescnt is
subtype COUNTER is UNSIGNED (1 downto 0);
signal COUNT: COUNTER;
constant ZERO: COUNTER := "00";
begin

process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then COUNT <= ZERO;
      else COUNT <= COUNT + ('0', X) + ('0', X);
      end if;
    end if;
  end process;

Z <= '1' when COUNT = ZERO else '0';

end Vonescnt_arch;
```

Table 9-18 Alternative VHDL process for ones-counting machine.

```
process (CLOCK)
variable ONES: STD_LOGIC_VECTOR (1 to 2);
begin
  if CLOCK'event and CLOCK = '1' then
    ONES := (X, Y);
    if RESET = '1' then COUNT <= ZERO;
    else case ONES is
           when "01" | "10" => COUNT <= COUNT + "01";
           when "11"        => COUNT <= COUNT + "10";
           when others      => null;
         end case;
    end if;
  end if;
end process;
```

Within the process, we use the usual method to check for a rising edge on `CLOCK`. The "`if`" clause performs a synchronous reset when required, and the "`else`" clause elegantly adds 0, 1 or 2 to `COUNT` depending on the values of `X` and `Y`. Recall that an expression such as "`('0', X)`" is an array literal; here we get an array of two `STD_LOGIC` elements, `'0'` and the current value of `X`. The type of this literal is compatible with `UNSIGNED`, since the number and type of elements are the same, so they can be combined using the "+" operation defined in the `std_logic_arith` package. Outside the process, the concurrent signal-assignment statement sets the Moore output `Z` to 1 when `COUNT` is zero.

For synthesis purposes, the "`if`" statement and assignment to `COUNT` in Table 9-17 doesn't necessarily yield a compact or speedy circuit. With a simple-minded synthesis tool, it could yield two 2-bit adders connected in series. Another approach is shown in Table 9-18. An intelligent tool may be able to synthesize a more compact incrementer for each of the two additions. In any case, formulating the choices in a case statement allows the two adders or incrementers to operate in parallel, and a multiplexer can be used to select one of their outputs according to the choices.

A final example for this subsection is the combination-lock state machine from Section 7.4 (below we omit the HINT output in the original specification):

> Design a clocked synchronous state machine with one input, X, and one output, UNLK. The UNLK output should be 1 if and only if X is 0 and the sequence of inputs received on X at the preceding seven clock ticks was 0110111.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vcomblck is
  port ( CLOCK, RESET, X: in STD_LOGIC;
         UNLK: out STD_LOGIC );
end;

architecture Vcomblck_arch of Vcomblck is
signal XHISTORY: STD_LOGIC_VECTOR (7 downto 1);
constant COMBINATION: STD_LOGIC_VECTOR (7 downto 1) := "0110111";
begin

  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then XHISTORY <= "0000000";
      else XHISTORY <= XHISTORY(6 downto 1) & X;
      end if;
    end if;
  end process;

  UNLK <= '1' when (XHISTORY=COMBINATION) and (X='0') else '0';

end Vcomblck_arch;
```

Table 9-19 VHDL program for finite-memory design of combination-lock state machine.

We developed a state table for this machine in Table 7-14 on page 582. But once again we can take a different approach that is easier to understand. Here, we note that the output of the machine at any time is completely determined by its inputs over the preceding eight clock ticks. Thus, we can use the so-called “finite-memory” approach to design this machine (see box on page 801). With this approach, we explicitly keep track of the past seven inputs and then form the output as a combinational function of these inputs.

The VHDL program in Table 9-19 is a finite-memory design. Within the architecture, the process merely keeps track of the last seven values of X using what’s essentially a shift register, bit 7 being the oldest value of X. (Recall that the “&” operator in VHDL is array concatenation.) Outside of the process, the concurrent signal-assignment statement sets the Mealy output UNLK to 1 when X is 0 and the 7-bit history matches the combination.

9.2.2 T-Bird Tail Lights

We described and designed a “T-bird tail-lights” state machine in Section 7.5. Table 9-20 is an equivalent VHDL program for the T-bird tail-lights machine. The state transitions in this machine are defined exactly the same as in the state diagram of Figure 7-64 on page 589. The machine uses an output-coded state assignment, taking advantage of the fact that the tail-light output values are different in each state.

Table 9-20 VHDL program for the T-bird tail-lights machine.

```
entity Vtbird is
  port ( CLOCK, RESET, LEFT, RIGHT, HAZ: in STD_LOGIC;
         LIGHTS: buffer STD_LOGIC_VECTOR (1 to 6) );
end;

architecture Vtbird_arch of Vtbird is
constant IDLE: STD_LOGIC_VECTOR (1 to 6) := "000000";
constant L3  : STD_LOGIC_VECTOR (1 to 6) := "111000";
constant L2  : STD_LOGIC_VECTOR (1 to 6) := "110000";
constant L1  : STD_LOGIC_VECTOR (1 to 6) := "100000";
constant R1  : STD_LOGIC_VECTOR (1 to 6) := "000001";
constant R2  : STD_LOGIC_VECTOR (1 to 6) := "000011";
constant R3  : STD_LOGIC_VECTOR (1 to 6) := "000111";
constant LR3 : STD_LOGIC_VECTOR (1 to 6) := "111111";
begin
  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then LIGHTS <= IDLE; else
        case LIGHTS is
          when IDLE => if HAZ='1' or (LEFT='1' and RIGHT='1') then LIGHTS <= LR3;
                       elsif LEFT='1'                          then LIGHTS <= L1;
                       elsif RIGHT='1'                         then LIGHTS <= R1;
                       else                                         LIGHTS <= IDLE;
                       end if;
          when L1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L2; end if;
          when L2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= L3; end if;
          when L3   => LIGHTS <= IDLE;
          when R1   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R2; end if;
          when R2   => if HAZ='1' then LIGHTS <= LR3; else LIGHTS <= R3; end if;
          when R3   => LIGHTS <= IDLE;
          when LR3  => LIGHTS <= IDLE;
          when others => null;
        end case;
      end if;
    end if;
  end process;
end Vtbird_arch;
```

IDLE MUSINGS In VHDL state machines, it's not necessary to make an explicit assignment of a next state if it's the same state that the machine is already in. In the execution of a process, a VHDL signal keeps its value if no assignment is made to it. Thus, in Table 9-20, the final "`else`" clause could be omitted in the `IDLE` state, with no effect on the machine's behavior.

Separately, the robustness of the state machine in Table 9-20 could be improved by replacing the "`null`" statement in the "`when others`" case with a transition to the `IDLE` state.

9.2.3 The Guessing Game

A "guessing game" machine was defined in Section 7.7.1 starting on page 594, with the following description:

> Design a clocked synchronous state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to lamps or LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red lamp. In normal operation, the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.
>
> Guesses are made by pressing a pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the "wrong" pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the lamp output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

As we discussed in Section 7.7.1, the machine requires six states—four in which a corresponding lamp is on, and two for when play is stopped after either a good or a bad pushbutton push. A VHDL program for the guessing game is shown in Table 9-21. This version also includes a `RESET` input that forces the game to a known starting state.

The program is pretty much a straightforward translation of the original state diagram in Figure 7-66 on page 597. Perhaps its only noteworthy feature is in the "`SOK | SERR`" case. Since the next-state transitions for these two states are identical (either go to `S1` or stay in the current state), they can be handled in one case. However, this tricky style of saving typing isn't particularly desirable from the point of view of state-machine documentation or maintainability. In the author's case, the trick's primary benefit was to help fit the program on one book page!

Table 9-21 VHDL program for the guessing-game machine.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vggame is
  port ( CLOCK, RESET, G1, G2, G3, G4: in  STD_LOGIC;
         L1, L2, L3, L4, ERR:          out STD_LOGIC );
end;

architecture Vggame_arch of Vggame is
type Sreg_type is (S1, S2, S3, S4, SOK, SERR);
signal Sreg: Sreg_type;
begin

  process (CLOCK)
  begin
    if CLOCK'event and CLOCK = '1' then
      if RESET = '1' then Sreg <= SOK; else
        case Sreg is
          when S1 => if    G2='1' or G3='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S2;
                     end if;
          when S2 => if    G1='1' or G3='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S3;
                     end if;
          when S3 => if    G1='1' or G2='1' or G4='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S4;
                     end if;
          when S4 => if    G1='1' or G2='1' or G3='1' then Sreg <= SERR;
                     elsif G1='1'                     then Sreg <= SOK;
                     else                                  Sreg <= S1;
                     end if;
          when SOK | SERR => if G1='0' and G2='0' and G3='0' and G4='0'
                          then Sreg <= S1; end if;
          when others => Sreg <= S1;
        end case;
      end if;
    end if;
  end process;

  L1  <= '1' when Sreg = S1   else '0';
  L2  <= '1' when Sreg = S2   else '0';
  L3  <= '1' when Sreg = S3   else '0';
  L4  <= '1' when Sreg = S4   else '0';
  ERR <= '1' when Sreg = SERR else '0';

end Vggame_arch;
```

The program in Table 9-21 does not specify a state assignment; a typical synthesis engine will use three bits for Sreg and assign the six states in order to binary combinations 000–101. For this state machine, it is also possible to use an output coded state assignment, using just the lamp and error output signals that are already required. VHDL does not provide a convenient mechanism for grouping together the entity's existing output signals and using them for state, but we can still achieve the desired effect with the changes shown in Table 9-22. Here we used a comment to document the correspondence between outputs and the bits of the new, 5-bit Sreg, and we changed each of the output assignment statements to pick off the appropriate bit instead of fully decoding the state.

Table 9-22 VHDL architecture for guessing game using output-coded state assignment.

```
architecture Vggameoc_arch of Vggame is
signal Sreg: STD_LOGIC_VECTOR (1 to 5);
-- bit positions of output-coded assignment: L1, L2, L3, L4, ERR
constant S1:   STD_LOGIC_VECTOR (1 to 5) := "10000";
constant S2:   STD_LOGIC_VECTOR (1 to 5) := "01000";
constant S3:   STD_LOGIC_VECTOR (1 to 5) := "00100";
constant S4:   STD_LOGIC_VECTOR (1 to 5) := "00010";
constant SERR: STD_LOGIC_VECTOR (1 to 5) := "00001";
constant SOK:  STD_LOGIC_VECTOR (1 to 5) := "00000";
begin

  process (CLOCK)
  ...                 (no change to process)
  end process;

  L1  <= Sreg(1);
  L2  <= Sreg(2);
  L3  <= Sreg(3);
  L4  <= Sreg(4);
  ERR <= Sreg(5);

end Vggameoc_arch;
```

9.2.4 Reinventing Traffic-Light Controllers

If you read the ABEL example in Section 9.1.5, then you've already heard me rant about the horrible traffic light controllers in Sunnyvale, California. They really *do* seem to be carefully designed to *maximize* the waiting time of cars at intersections. In this section we'll design a traffic-light controller with distinctly Sunnyvale-like behavior.

An infrequently used intersection (one that would have no more than a "yield" sign if it were in Chicago) has the sensors and signals shown in

Table 9-23 VHDL program for Sunnyvale traffic-light controller.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG: in  STD_LOGIC;
         OVERRIDE, FLASHCLK:                          in  STD_LOGIC;
         NSRED, NSYELLOW, NSGREEN:                    out STD_LOGIC;
         EWRED, EWYELLOW, EWGREEN, TMRESET:           out STD_LOGIC );
end;

architecture Vsvale_arch of Vsvale is
type Sreg_type is (NSGO, NSWAIT, NSWAIT2, NSDELAY,
                   EWGO, EWWAIT, EWWAIT2, EWDELAY);
signal Sreg: Sreg_type;
begin

process (CLOCK)
begin
  if CLOCK'event and CLOCK = '1' then
    if RESET = '1' then Sreg <= NSDELAY; else
      case Sreg is
        when NSGO =>                                       -- North-south green.
          if    TMSHORT='0' then Sreg <= NSGO;             -- Minimum 5 seconds.
          elsif TMLONG='1'  then Sreg <= NSWAIT;           -- Maximum 5 minutes.
          elsif EWCAR='1' and NSCAR='0' then Sreg <= NSGO;   -- Make EW car wait.
          elsif EWCAR='1' and NSCAR='1' then Sreg <= NSWAIT; -- Thrash if cars both ways.
          elsif EWCAR='0' and NSCAR='1' then Sreg <= NSWAIT; -- New NS car? Make it stop!
          else                               Sreg <= NSGO;   -- No one coming, no change.
          end if;
        when NSWAIT  => Sreg <= NSWAIT2;                   -- Yellow light,
        when NSWAIT2 => Sreg <= NSDELAY;                   --   two ticks for safety.
        when NSDELAY => Sreg <= EWGO;                      -- Red both ways for safety.
        when EWGO =>                                       -- East-west green.
          if    TMSHORT='0' then Sreg <= EWGO;             -- Same behavior as above.
          elsif TMLONG='1'  then Sreg <= EWWAIT;
          elsif NSCAR='1' and EWCAR='0' then Sreg <= EWGO;
          elsif NSCAR='1' and EWCAR='1' then Sreg <= EWWAIT;
          elsif NSCAR='0' and EWCAR='1' then Sreg <= EWWAIT;
          else                               Sreg <= EWGO;
          end if;
        when EWWAIT  => Sreg <= EWWAIT2;
        when EWWAIT2 => Sreg <= EWDELAY;
        when EWDELAY => Sreg <= NSGO;
        when others  => Sreg <= NSDELAY;                   -- "Reset" state.
      end case;
    end if;
  end if;
end process;
```

Figure 9-5 on page 809. The state machine that controls the traffic signals uses a 1 Hz clock and a timer and has four inputs:

NSCAR	Asserted when a car on the north-south road is over either sensor on either side of the intersection.
EWCAR	Asserted when a car on the east-west road is over either sensor on either side of the intersection.
TMLONG	Asserted if more than five minutes has elapsed since the timer started; remains asserted until the timer is reset.
TMSHORT	Asserted if more than five seconds has elapsed since the timer started; remains asserted until the timer is reset.

The state machine has seven outputs:

NSRED, NSYELLOW, NSGREEN	Control the north-south lights.
EWRED, EWYELLOW, EWGREEN	Control the east-west lights.
TMRESET	When asserted, resets the timer and negates TMSHORT and TMLONG. The timer starts timing when TMRESET is negated.

A typical, municipally approved algorithm for controlling the traffic lights is embedded in the VHDL program of Table 9-23. This algorithm produces two frequently seen behaviors of “smart” traffic lights. At night, when traffic is light, it holds a car stopped at the light for up to five minutes, unless a car approaches on the cross street, in which case it stops the cross traffic and lets the waiting car go. (The “early warning” sensor is far enough back to change the lights before the approaching car reaches the intersection.) During the day, when traffic is heavy and there are always cars waiting in both directions, it cycles the lights every five seconds, thus minimizing the utilization of the intersection and maximizing everyone’s waiting time, thereby creating a public outcry for more taxes to fix the problem.

Table 9-23 (continued) VHDL program for Sunnyvale traffic-light controller.

```
TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=NSGO and Sreg/=NSWAIT and Sreg/=NSWAIT2 else '0';
NSYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=NSWAIT or Sreg=NSWAIT2 else '0';
NSGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=NSGO else '0';
EWRED    <= FLASHCLK when OVERRIDE='1' else
            '1' when Sreg/=EWGO and Sreg/=EWWAIT and Sreg/=EWWAIT2 else '0';
EWYELLOW <= '0' when OVERRIDE='1' else
            '1' when Sreg=EWWAIT or Sreg=EWWAIT2 else '0';
EWGREEN  <= '0' when OVERRIDE='1' else '1' when Sreg=EWGO else '0';

end Vsvale_arch;
```

While writing the program, we took the opportunity to add two inputs that weren't in the original specification. The OVERRIDE input may be asserted by the police to disable the controller and put the signals into a flashing-red mode at a rate determined by the FLASHCLK input. This allows them to manually clear up the traffic snarls created by this wonderful invention.

Table 9-24 Definitions for Sunnyvale traffic-lights machine with output-coded state assignment.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Vsvale is
  port ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG: in  STD_LOGIC;
         OVERRIDE, FLASHCLK:                          in  STD_LOGIC;
         NSRED, NSYELLOW, NSGREEN:                    out STD_LOGIC;
         EWRED, EWYELLOW, EWGREEN, TMRESET:           out STD_LOGIC );
end;

architecture Vsvaleoc_arch of Vsvale is
signal Sreg: STD_LOGIC_VECTOR (1 to 7);
-- bit positions of output-coded assignment:  (1) NSRED, (2) NSYELLOW, (3) NSGREEN,
--                                    (4) EWRED, (5) EWYELLOW, (6) EWGREEN, (7) EXTRA
constant NSGO:    STD_LOGIC_VECTOR (1 to 7) := "0011000";
constant NSWAIT:  STD_LOGIC_VECTOR (1 to 7) := "0101000";
constant NSWAIT2: STD_LOGIC_VECTOR (1 to 7) := "0101001";
constant NSDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001000";
constant EWGO:    STD_LOGIC_VECTOR (1 to 7) := "1000010";
constant EWWAIT:  STD_LOGIC_VECTOR (1 to 7) := "1000100";
constant EWWAIT2: STD_LOGIC_VECTOR (1 to 7) := "1000101";
constant EWDELAY: STD_LOGIC_VECTOR (1 to 7) := "1001001";

begin

process (CLOCK)
...                (no change to process)
end process;

TMRESET  <= '1' when Sreg=NSWAIT2 or Sreg=EWWAIT2 else '0';
NSRED    <= Sreg(1);
NSYELLOW <= Sreg(2);
NSGREEN  <= Sreg(3);
EWRED    <= Sreg(4);
EWYELLOW <= Sreg(5);
EWGREEN  <= Sreg(6);

end Vsvaleoc_arch;
```

Like most of our other examples, Table 9-23 does not give a specific state assignment. And like many of our other examples, this state machine works well with an output-coded state assignment. Many of the states can be identified by a unique combination of light-output values. But there are three pairs of states that are not distinguishable by looking at the lights alone: (NSWAIT, NSWAIT2), (EWWAIT, EWWAIT2), and (NSDELAY, EQDELAY). We can handle these by adding one more state variable, “EXTRA”, that has different values for the two states in each pair. This idea is realized in the modified program in Table 9-24.

Exercises

9.1 Write an ABEL program for the state machine described in Exercise 7.30.

9.2 Modify the ABEL program of Table 9-2 to include the HINT output from the original state-machine specification in Section 7.4.

9.3 Redesign the T-bird tail-lights machine of Section 9.1.3 to include parking-light and brake-light functions. When the BRAKE input is asserted, all of the lights should go on immediately, and stay on until BRAKE is negated, independent of any other function. When the PARK input is asserted, each lamp is turned on at 50% brightness at all times when it would otherwise be off. This is achieved by driving the lamp with a 100 Hz signal DIMCLK with a 50% duty cycle. Draw a logic diagram for the circuit using one or two PLDs, write an ABEL program for each PLD, and write a short description of how your system works.

9.4 Find a 3-bit state assignment for the guessing-game machine in Table 9-5 that reduces the maximum number of product terms per output to 7. Can you do even better?

9.5 The operation of the guessing game in Section 9.1.4 is very predictable; it’s easy for a player to learn the rate at which the lights change and always hit the button at the right time. The game is more fun if the rate of change is more variable.

Modify the ABEL state machine in Table 9-5 so that in states S1–S4, the machine advances occurs only if a new input, SEN, is asserted. (SEN is intended to be hooked up to a pseudo-random bit-stream generator.) Both correct and incorrect button pushs should be recognized whether or not SEN is asserted. Determine whether your modified design still fits in a 16V8.

9.6 In connection with the preceding exercise, write an ABEL program for an 8-bit LFSR using a single 16V8, such that one of its outputs can be used as a pseudo-random bit-stream generator. After how many clock ticks does the bit sequence repeat? What is the maximum number of 0s that occur in a row? of 1s?

9.7 Add an OVERRIDE input to the traffic-lights state machine of Figure 9-7, still using just a single 16V8. When OVERRIDE is asserted, the red lights should flash on and off, changing once per second. Write a complete ABEL program for your machine.

9.8 Modify the behavior of the ABEL traffic-light-controller machine in Table 9-9 to have more reasonable behavior, the kind you’d like to see for traffic lights in your own home town.

9.9 Write a VHDL program for the state machine described in Exercise 7.30.

9.10 Show how to modify Table 9-13 to provide an asynchronous RESET input that forces the state machine to the INIT state. Repeat for a synchronous version that forces the state machine to the INIT state if RESET is asserted on the rising clock edge.

9.11 Write and test a VHDL program corresponding to Figure 9-8(b). Using an available synthesis tool, determine whether the circuit resulting from this version differs from the one resulting from Figure 9-8(a), and how.

9.12 Write a VHDL program for the ones-counting state machine as described by the state table in Table 7-12.

9.13 Write a VHDL program for the combination-lock state machine as described by the state table in Table 7-14.

9.14 Modify the VHDL program of Table 9-19 to include the HINT output from the original state-machine specification in Section 7.4.

9.15 Repeat Exercise 9.3 using VHDL, assuming you are targeting your design to a single CPLD or FPGA.

9.16 Modify the VHDL guessing-game program of Table 9-21 according to the idea in Exercise 9.5. Add another process to the program to provide a pseudo-random bit stream generator according to Exercise 9.6.

9.17 Modify the behavior of the VHDL traffic-light-controller machine in Table 9-23 to have more reasonable behavior, the kind you'd like to see for traffic lights in your own home town.