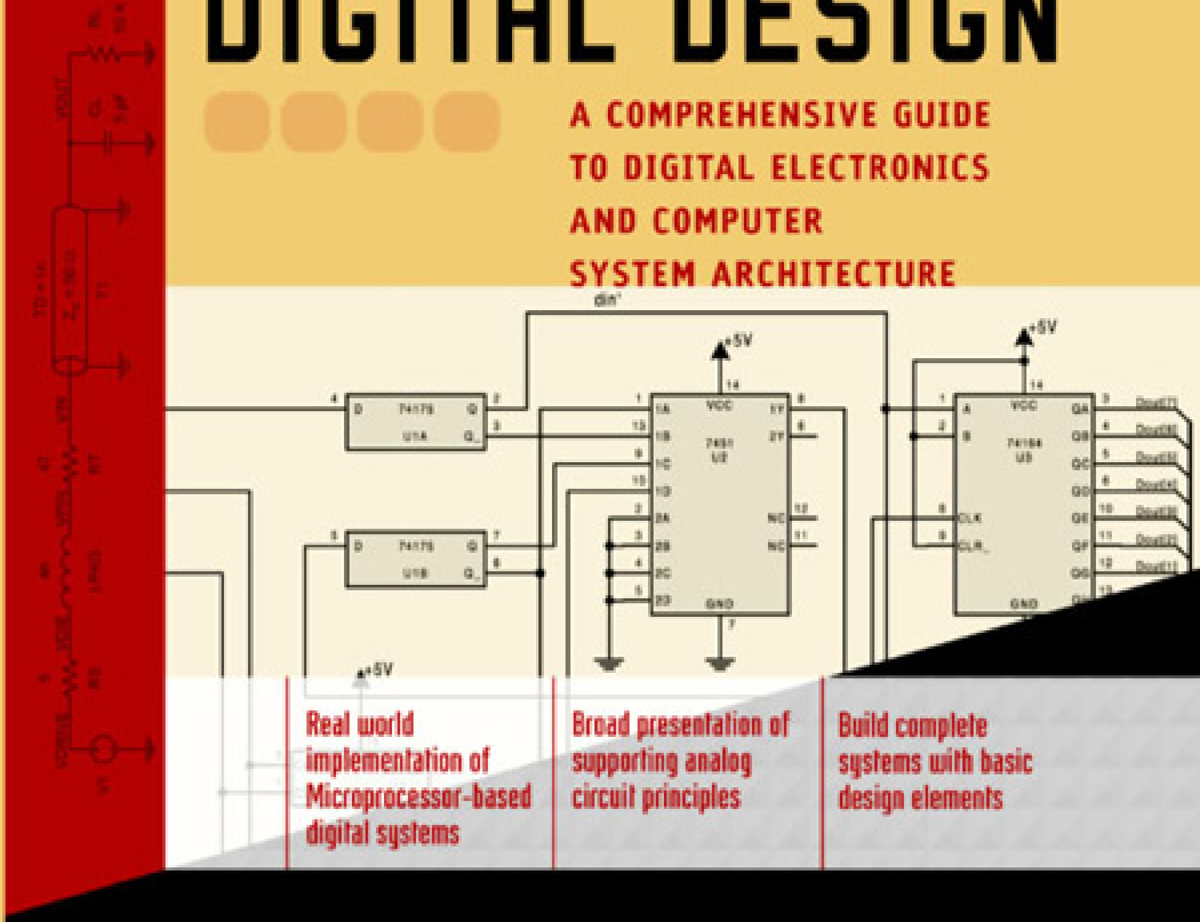
COMPLETE DIGITAL DESIGN

A COMPREHENSIVE GUIDE TO DIGITAL ELECTRONICS AND COMPUTER SYSTEM ARCHITECTURE



Real world implementation of Microprocessor-based digital systems

Broad presentation of supporting analog circuit principles

Build complete systems with basic design elements

MARK BALCH

COMPLETE DIGITAL DESIGN

Chapter 5, "Serial Communications," presents one of the most basic aspects of systems design: moving data from one system to another. Without data links, computers would be isolated islands. Communication is key to many applications, whether accessing the Internet or gathering data from a remote sensor. Topics including RS-232 interfaces, modems, and basic multinode networking are discussed with a focus on implementing real data links.

Chapter 6, "Instructive Microprocessors and Microcomputer Elements," walks through five examples of real microprocessors and microcontrollers. The devices presented are significant because of their trail-blazing roles in defining modern computing architecture, as exhibited by the fact that, decades later, they continue to turn up in new designs in one form or another. These devices are used as vehicles to explain a wide range of computing issues from register, memory, and bus architectures to interrupt vectoring and operating system privilege levels.

PART 2 ADVANCED DIGITAL SYSTEMS

Digital systems operate by acquiring data, manipulating that data, and then transferring the results as dictated by the application. Part 2 builds on the foundations of Part 1 by exploring the state of the art in microprocessor, memory, communications, and logic implementation technologies. To effectively conceive and implement such systems requires an understanding of what is possible, what is practical, and what tools and building blocks exist with which to get started. On completing Parts 1 and 2, you will have acquired a broad understanding of digital systems ranging from small microcontrollers to 32-bit microcomputer architecture and high-speed networking, and the logic design methodologies that underlie them all. You will have the ability to look at a digital system, whether pre-existing or conceptual, and break it into its component parts—the first step in solving a problem.

Chapter 7, "Advanced Microprocessor Concepts," discusses the key architectural topics behind modern 32- and 64-bit computing systems. Basic concepts including RISC/CISC, floating-point arithmetic, caching, virtual memory, pipelining, and DSP are presented from the perspective of what a digital hardware engineer needs to know to understand system-wide implications and design useful circuits. This chapter does not instruct the reader on how to build the fastest microprocessors, but it does explain how these devices operate and, more importantly, what system-level design considerations and resources are necessary to achieve a functioning system.

Chapter 8, "High-Performance Memory Technologies," presents the latest SDR/DDR SDRAM and SDR/DDR/QDR SSRAM devices, explains how they work and why they are useful in high-performance digital systems, and discusses the design implications of each. Memory is used by more than just microprocessors. Memory is essential to communications and data processing systems. Understanding the capabilities and trade-offs of such a central set of technologies is crucial to designing a practical system. Familiarity with all mainstream memory technologies is provided to enable a firm grasp of the applications best suited to each.

Chapter 9, "Networking," covers the broad field of digital communications from a digital hardware perspective. Network protocol layering is introduced to explain the various levels at which

Chapter 15, “Analog Interfaces for Digital Systems,” covers the basics of analog-to-digital and digital-to-analog conversion techniques. Many digital systems interact with real-world stimuli including audio, video, and radio frequencies. Data conversion is a key portion of these systems, enabling continuous analog signals to be represented and processed as binary numbers. Several common means of performing data conversion are discussed along with fundamental concepts such as the Nyquist frequency and anti-alias filtering.

PART 4 DIGITAL SYSTEM DESIGN IN PRACTICE

When starting to design a new digital system, high-profile features such as the microprocessor and memory architecture often get most of the attention. Yet there are essential support elements that may be overlooked by those unfamiliar with them and unaware of the consequences of not taking time to address necessary details. All too often, digital engineers end up with systems that almost work. A microprocessor may work properly for a few hours and then quit. A data link may work fine one day and then experience inexplicable bit errors the next day. Sometimes these problems are the result of logic bugs, but mysterious behavior may be related to a more fundamental electrical flaw. The final part of this book explains the supporting infrastructure and electrical phenomena that must be understood to design and build reliable systems.

Chapter 16, “Clock Distribution,” explores an essential component of all digital systems: proper generation and distribution of clocks. Many common clock generation and distribution methods are presented with detailed circuit implementation examples including low-skew buffers, termination, and PLLs. Related subjects, including frequency synthesis, DLLs, and source-synchronous clocking, are presented to lend a broad perspective on system-level clocking strategies.

Chapter 17, “Voltage Regulation and Power Distribution” discusses the fundamental power infrastructure necessary for system operation. An introduction to general power handling is provided that covers issues such as circuit specifications and safety issues. Thermal analysis is emphasized for safety and reliability concerns. Basic regulator design with discrete components and integrated circuits is explained with numerous illustrative circuits for each topic. The remainder of the chapter addresses power distribution topics including wiring, circuit board power planes, and power supply decoupling capacitors.

Chapter 18, “Signal Integrity,” delves into a set of topics that addresses the nonideal behavior of high-speed digital signals. The first half of this chapter covers phenomena that are common causes of corrupted digital signals. Transmission lines, signal reflections, crosstalk, and a wide variety of termination schemes are explained. These topics provide a basic understanding of what can go wrong and how circuits and systems can be designed to avoid signal integrity problems. Electromagnetic radiation, grounding, and static discharge are closely related subjects that are presented in the second half of the chapter. An overview is presented of the problems that can arise and their possible solutions. Examples illustrate concepts that apply to both circuit board design and overall system enclosure design—two equally important matters for consideration.

Chapter 19, “Designing for Success,” explores a wide range of system-level considerations that should be taken into account during the product definition and design phases of a project. Component selection and circuit fabrication must complement the product requirements and available development and manufacturing resources. Often considered mundane, these topics are discussed because a successful outcome hinges on the availability and practicality of parts and technologies that are designed into a system. System testability is emphasized in this chapter from several perspectives, because testing is prominent in several phases of product development. Test mechanisms including boundary scan (JTAG), specific hardware features, and software diagnostic routines enable more efficient debugging and fault isolation in both laboratory and assembly line environments. Common computer-aided design software for digital systems is presented with an emphasis on Spice

a
s

g
tes.
es and
cts than
cturers not
ndiscovered.
t of the discus-
s data sheet, be-
tributes, and such
contact the manufac-
nponent manufacturers
se their products in a safe
they offer. The widespread

no substitute for experience or
this experience by being pointed
s, it can make it more enjoyable as
be more effectively channeled through
here to look for necessary information. I
n you the best of luck in your endeavors.

Mark Balch

Boolean variables may not seem too interesting on their own. It is what they can be made to represent that leads to useful constructs. A rather contrived example can be made from the following logical statement:

"If today is Saturday or Sunday and it is warm, then put on shorts."

Three Boolean inputs can be inferred from this statement: Saturday, Sunday, and warm. One Boolean output can be inferred: shorts. These four variables can be assembled into a single logic equation that computes the desired result,

shorts = (Saturday OR Sunday) AND warm

While this is a simple example, it is representative of the fact that any logical relationship can be expressed algebraically with products and sums by combining the basic logic functions AND, OR, and NOT.

Several other logic functions are regarded as elemental, even though they can be broken down into AND, OR, and NOT functions. These are not–AND (NAND), not–OR (NOR), exclusive–OR (XOR), and exclusive–NOR (XNOR). Table 1.3 presents the logical definitions of these other basic functions. XOR is an interesting function, because it implements a sum that is distinct from OR by taking into account that 1 + 1 does not equal 1. As will be seen later, XOR plays a key role in arithmetic for this reason.

TABLE 1.3 NAND, NOR, XOR, XNOR Truth Table

A	B	A NAND B	A NOR B	A XOR B	A XNOR B
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

All binary operators can be chained together to implement a wide function of any number of inputs. For example, the truth table for a ten-input AND function would result in a 1 output only when all inputs are 1. Similarly, the truth table for a seven-input OR function would result in a 1 output if any of the seven inputs are 1. A four-input XOR, however, will only result in a 1 output if there are an odd number of ones at the inputs. This is because of the logical daisy chaining of multiple binary XOR operations. As shown in Table 1.3, an even number of 1s presented to an XOR function cancel each other out.

It quickly grows unwieldy to write out the names of logical operators. Concise algebraic expressions are written by using the graphical representations shown in Table 1.4. Note that each operation has multiple symbolic representations. The choice of representation is a matter of style when handwritten and is predetermined when programming a computer by the syntactical requirements of each computer programming language.

A common means of representing the output of a generic logical function is with the variable Y. Therefore, the AND function of two variables, A and B, can be written as Y = A & B or Y = A*B. As with normal mathematical notation, products can also be written by placing terms right next to each other, such as Y = AB. Notation for the inverted functions, NAND, NOR, and XNOR, is achieved by

TABLE 1.4 Symbolic Representations of Standard Boolean Operators

Boolean Operation	Operators
AND	*, &
OR	+, |, #
XOR	⊕, ^
NOT	!, ~, $\overline{\text{A}}$

inverting the base function. Two equally valid ways of representing NAND are Y = $\overline{\text{A \& B}}$ and Y = !(AB). Similarly, an XNOR might be written as Y = $\overline{\text{A} \oplus \text{B}}$.

When logical functions are converted into circuits, graphical representations of the seven basic operators are commonly used. In circuit terminology, the logical operators are called *gates*. Figure 1.1 shows how the basic logic gates are drawn on a circuit diagram. Naming the inputs of each gate A and B and the output Y is for reference only; any name can be chosen for convenience. A small bubble is drawn at a gate's output to indicate a logical inversion.

More complex Boolean functions are created by combining Boolean operators in the same way that arithmetic operators are combined in normal mathematics. Parentheses are useful to explicitly convey precedence information so that there is no ambiguity over how two variables should be treated. A Boolean function might be written as

$$Y = (AB + \overline{C} + D)\&\overline{E \oplus F}$$

This same equation could be represented graphically in a circuit diagram, also called a *schematic diagram*, as shown in Fig. 1.2. This representation uses only two-input logic gates. As already mentioned, binary operators can be chained together to implement functions of more than two variables.

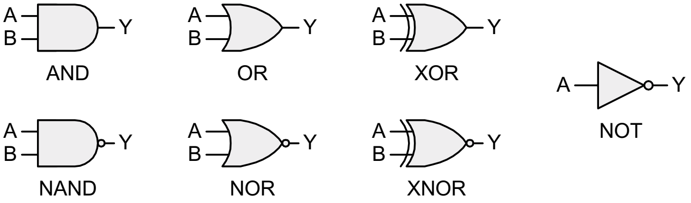


FIGURE 1.1 Graphical representation of basic logic gates.

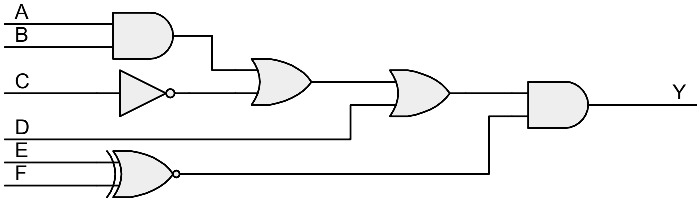


FIGURE 1.2 Schematic diagram of logic function.

An alternative graphical representation would use a three-input OR gate by collapsing the two-input OR gates into a single entity.

1.2 BOOLEAN MANIPULATION

Boolean equations are invaluable when designing digital logic. To properly use and devise such equations, it is helpful to understand certain basic rules that enable simplification and re-expression of Boolean logic. Simplification is perhaps the most practical final result of Boolean manipulation, because it is easier and less expensive to build a circuit that does not contain unnecessary components. When a logical relationship is first set down on paper, it often is not in its most simplified form. Such a circuit will function but may be unnecessarily complex. Re-expression of a Boolean equation is a useful skill, because it can enable you to take better advantage of the logic resources at your disposal instead of always having to use new components each time the logic is expanded or otherwise modified to function in a different manner. As will soon be shown, an OR gate can be made to behave as an AND gate, and vice versa. Such knowledge can enable you to build a less-complex implementation of a Boolean equation.

First, it is useful to mention two basic identities:

$$A \mathbin{\&} \overline{A} = 0 \text{ and } A + \overline{A} = 1$$

The first identity states that the product of any variable and its logical negation must always be false. It has already been shown that both operands of an AND function must be true for the result to be true. Therefore, the first identity holds true, because it is impossible for both operands to be true when one is the negation of the other. The second identity states that the sum of any variable and its logical negation must always be true. At least one operand of an OR function must be true for the result to be true. As with the first identity, it is guaranteed that one operand will be true, and the other will be false.

Boolean algebra also has commutative, associative, and distributive properties as listed below:

- Commutative: $A \mathbin{\&} B = B \mathbin{\&} A$ and $A + B = B + A$
- Associative: $(A \mathbin{\&} B) \mathbin{\&} C = A \mathbin{\&} (B \mathbin{\&} C)$ and $(A + B) + C = A + (B + C)$
- Distributive: $A \mathbin{\&} (B + C) = A \mathbin{\&} B + A \mathbin{\&} C$

The aforementioned identities, combined with these basic properties, can be used to simplify logic. For example,

$$A \mathbin{\&} B \mathbin{\&} C + A \mathbin{\&} B \mathbin{\&} \overline{C}$$

can be re-expressed using the distributive property as

$$A \mathbin{\&} B \mathbin{\&} (C + \overline{C})$$

which we know by identity equals

$$A \mathbin{\&} B \mathbin{\&} (1) = A \mathbin{\&} B$$

Another useful identity, $A + \overline{A}B = A + B$, can be illustrated using the truth table shown in Table 1.5.

If the corresponding Boolean equation does not immediately become clear, the truth table can be converted into a K-map as shown in Fig. 1.3. The K-map has one box for every combination of inputs, and the desired output for a given combination is written into the corresponding box. Each axis of a K-map represents up to two variables, enabling a K-map to solve a function of up to four variables. Individual grid locations on each axis are labeled with a unique combination of the variables represented on that axis. The labeling pattern is important, because only one variable per axis is permitted to differ between adjacent boxes. Therefore, the pattern "00, 01, 10, 11" is not proper, but the pattern "11, 01, 00, 10" would work as well as the pattern shown.

K-maps are solved using the *sum of products* principle, which states that any relationship can be expressed by the logical OR of one or more AND terms. Product terms in a K-map are recognized by picking out groups of adjacent boxes that all have a state of 1. The simplest product term is a single box with a 1 in it, and that term is the product of all variables in the K-map with each variable either inverted or not inverted such that the result is 1. For example, a 1 is observed in the box that corresponds to A = 0, B = 1, and C = 1. The product term representation of that box would be $\overline{A}BC$. A brute force solution is to sum together as many product terms as there are boxes with a state of 1 (there are five in this example) and then simplify the resulting equation to obtain the final result. This approach can be taken without going to the trouble of drawing a K-map. The purpose of a K-map is to help in identifying minimized product terms so that lengthy simplification steps are unnecessary.

Minimized product terms are identified by grouping together as many adjacent boxes with a state of 1 as possible, subject to the rules of Boolean algebra. Keep in mind that, to generate a valid product term, all boxes in a group must have an identical relationship to all of the equation's input variables. This requirement translates into a rule that product term groups must be found in power-of-two quantities. For a three-variable K-map, product term groups can have only 1, 2, 4, or 8 boxes in them.

Going back to our example, a four-box product term is formed by grouping together the vertically stacked 1s on the left and right edges of the K-map. An interesting aspect of a K-map is that an edge wraps around to the other side, because the axis labeling pattern remains continuous. The validity of this wrapping concept is shown by the fact that all four boxes share a common relationship with the input variables: their product term is $\overline{B}$. The other variables, A and C, can be ruled out, because the boxes are 1 regardless of the state of A and C. Only variable B is a determining factor, and it must be 0 for the boxes to have a state of 1. Once a product term has been identified, it is marked by drawing a ring around it as shown in Fig. 1.4. Because the product term crosses the edges of the table, half-rings are shown in the appropriate locations.

There is still a box with a 1 in it that has not yet been accounted for. One approach could be to generate a product term for that single box, but this would not result in a fully simplified equation, because a larger group can be formed by associating the lone box with the adjacent box corresponding to A = 0, B = 0, and C = 1. K-map boxes can be part of multiple groups, and forming the largest groups possible results in a fully simplified equation. This second group of boxes is circled in Fig. 1.5 to complete the map. This product term shares a common relationship where A = 0, C = 1, and B

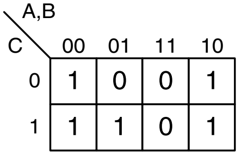


FIGURE 1.3 Karnaugh map for function of three variables.

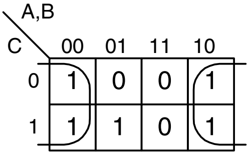


FIGURE 1.4 Partially completed Karnaugh map for a function of three variables.

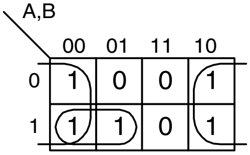
is irrelevant: $\overline{A}C$. It may appear tempting to create a product term consisting of the three boxes on the bottom edge of the K-map. This is not valid because it does not result in all boxes sharing a common product relationship, and therefore violates the power-of-two rule mentioned previously. Upon completing the K-map, all product terms are summed to yield a final and simplified Boolean equation that relates the input variables and the output: .

Functions of four variables are just as easy to solve using a K-map. Beyond four variables, it is preferable to break complex functions into smaller subfunctions and then combine the Boolean equations once they have been determined. Figure 1.6 shows an example of a completed Karnaugh map for a hypothetical function of four variables. Note the overlap between several groups to achieve a simplified set of product terms. The lager a group is, the fewer unique terms will be required to represent its logic. There is nothing to lose and something to gain by forming a larger group whenever possible. This K-map has four product terms that are summed for a final result: .

In both preceding examples, each result box in the truth table and Karnaugh map had a clearly defined state. Some logical relationships, however, do not require that every possible result necessarily be a one or a zero. For example, out of 16 possible results from the combination of four variables, only 14 results may be mandated by the application. This may sound odd, but one explanation could be that the particular application simply cannot provide the full 16 combinations of inputs. The specific reasons for this are as numerous as the many different applications that exist. In such circumstances these so-called *don't care* results can be used to reduce the complexity of your logic. Because the application does not care what result is generated for these few combinations, you can arbitrarily set the results to 0s or 1s so that the logic is minimized. Figure 1.7 is an example that modifies the Karnaugh map in Fig. 1.6 such that two don't care boxes are present. Don't care values are most commonly represented with "x" characters. The presence of one x enables simplification of the resulting logic by converting it to a 1 and grouping it with an adjacent 1. The other x is set to 0 so that it does not waste additional logic terms. The new Boolean equation is simplified by removing B from the last term, yielding . It is helpful to remember that x values can generally work to your benefit, because their presence imposes fewer requirements on the logic that you must create to get the job done.

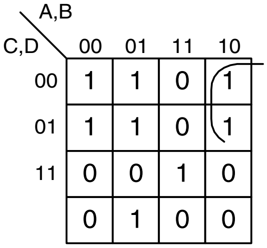
1.4 BINARY AND HEXADECIMAL NUMBERING

The fact that there are only two valid Boolean values, 1 and 0, makes the *binary* numbering system appropriate for logical expression and, therefore, for digital systems. Binary is a base-2 system in



A,B	00	01	11	10
0	1	0	0	1
1	1	1	0	1

FIGURE 1.5 Completed Karnaugh map for a function of three variables.



C,D \ A,B	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	0	0	1	0
	0	1	0	0

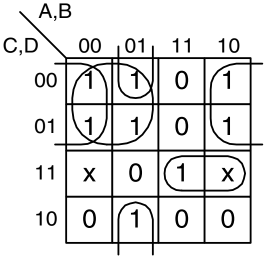


FIGURE 1.7 Karnaugh map for function of four variables with two "don't care" values.

which only the digits 1 and 0 exist. Binary follows the same laws of mathematics as decimal, or base-10, numbering. In decimal, the number 191 is understood to mean one hundreds plus nine tens plus one ones. It has this meaning, because each digit represents a successively higher power of ten as it moves farther left of the decimal point. Representing 191 in mathematical terms to illustrate these increasing powers of ten can be done as follows:

$$191 = 1 \times 10^2 + 9 \times 10^1 + 1 \times 10^0$$

Binary follows the same rule, but instead of powers of ten, it works on powers of two. The num-
10 in binary (written as 110_2 to explicitly denote base 2) does not equal 110_{10} (decimal).
$110_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{10}$. The number 191_{10} can be converted to binary by per-
uccessive division by decreasing powers of 2 as shown below:

$191 \div 2^7 \quad = 191 \div 128 \quad = 1$ remainder 63

$63 \div 2^6 \quad = 63 \div 64 \quad = 0$ remainder 63

$63 \div 2^5 \quad = 63 \div 32 \quad = 1$ remainder 31

$\div 2^4 \quad = 31 \div 16 \quad = 1$ remainder 15

$= 15 \div 8 \quad = 1$ remainder 7

$= 7 \div 4 \quad = 1$ remainder 3

$3 \div 2 \quad = 1$ remainder 1

$1 \quad = 1$ remainder 0

1_2. Each binary digit is referred to as a *bit*. A group of N
to $2^N - 1$. There are eight bits in a *byte*, more formally
te to represent numbers up to $2^8 - 1 = 255$. The pre-
rms in a byte. If each term, or bit, has its maximum
$+ 2 + 1 = 255$.
logic states, it is rather cumbersome to work
f ones and zeroes. *Hexadecimal*, or base 16
inary numbers in a more succinct notation.
it represents four binary digits, given that

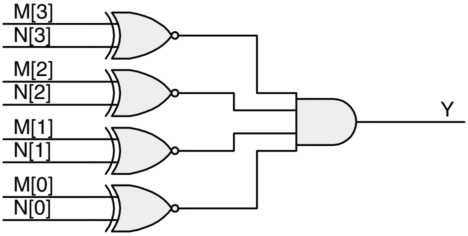


FIGURE 1.8 Four-bit equality logic.

Logic to compare one number against a constant is simpler than comparing two numbers, because the number of inputs to the Boolean equation is cut in half. If, for example, one wanted to compare M[3:0] to a constant 1001_2 (9_{10}), the logic would reduce to just a four-input AND gate with two inverted inputs:

$$y = M[3]\&\overline{M[2]}\&\overline{M[1]}\&M[0]$$

When working with computers and other digital systems, numbers are almost always written in hex notation simply because it is far easier to work with fewer digits. In a 32-bit computer, a value can be written as either 8 hex digits or 32 bits. The computer's logic always operates on raw binary quantities, but people generally find it easier to work in hex. An interesting historical note is that hex was not always the common method of choice for representing bits. In the early days of computing, through the 1960s and 1970s, *octal* (base-8) was used predominantly. Instead of a single hex digit representing four bits, a single octal digit represents three bits, because $2^3 = 8$. In octal, $191_{10} = 277_8$. Whereas bytes are the *lingua franca* of modern computing, groups of two or three octal digits were common in earlier times.

Because of the inherent binary nature of digital systems, quantities are most often expressed in orders of magnitude that are tied to binary rather than decimal numbering. For example, a "round number" of bytes would be 1,024 (2^{10}) rather than 1000 (10^3). Succinct terminology in reference to quantities of data is enabled by a set of standard prefixes used to denote order of magnitude. Furthermore, there is a convention of using a capital B to represent a quantity of bytes and using a lowercase b to represent a quantity of bits. Commonly observed prefixes used to quantify sets of data are listed in Table 1.8. Many memory chips and communications interfaces are expressed in units of bits. One must be careful not to misunderstand a specification. If you need to store 32 MB of data, be sure to use a 256 Mb memory chip rather than a 32 Mb device!

TABLE 1.8 Common Binary Magnitude Prefixes

Prefix	Definition	Order of Magnitude	Abbreviation	Usage
Kilo	$(1{,}024)^1 = 1{,}024$	2^{10}	k	kB
Mega	$(1{,}024)^2 = 1{,}048{,}576$	2^{20}	M	MB
Giga	$(1{,}024)^3 = 1{,}073{,}741{,}824$	2^{30}	G	GB
Tera	$(1{,}024)^4 = 1{,}099{,}511{,}627{,}776$	2^{40}	T	TB
Peta	$(1{,}024)^5 = 1{,}125{,}899{,}906{,}842{,}624$	2^{50}	P	PB
Exa	$(1{,}024)^6 = 1{,}152{,}921{,}504{,}606{,}846{,}976$	2^{60}	E	EB

alread
nary a
two num
any column
the next column. In binary, the la
The addition of 111_2 and 011_2 (7 +

			1	*0*	*carry bits*
		1	1	1	
+		0	1	1	
	1	0	1	0	

In the first column, the sum of two ues
The sum of the second column is 3 , or
in the sum. When all three colum ar
fourth column. The carry is, in fect

The logic to perform bina add
is the XOR gate, whose re ult is
generates a 1 when eith inpu
1, and 1 + 0. The fou n pos
carry is generated ly wh
called *half-adde* s repr

Th logic is
added togeth
ing from th
A *full-add*
tion circ
put rela
the ne
comb

rather than directly subtracting the subtrahend from the minuend. These are, of course, identical operations: A – B = A + (–B). This type of arithmetic is referred to as subtraction by addition of the *two's complement*. The two's complement is the negative representation of a number that allows the identity A – B = A + (–B) to hold true.

Subtraction requires a means of expressing negative numbers. To this end, the most-significant bit, or left-most bit, of a binary number is used as the sign-bit when dealing with signed numbers. A negative number is indicated when the sign-bit equals 1. Unsigned arithmetic does not involve a sign-bit, and therefore can express larger absolute numbers, because the MSB is merely an extra digit rather than a sign indicator.

The first step in performing two's complement subtraction is to convert the subtrahend into a negative equivalent. This conversion is a two-step process. First, the binary number is inverted to yield a *one's complement*. Then, 1 is added to the one's complement version to yield the desired two's complement number. This is illustrated below:

	0	1	0	1	Original number (5)
	1	0	1	0	One's complement
+	0	0	0	1	Add one
	1	0	1	1	Two's complement (–5)

Observe that the unsigned four-bit number that can represent values from 0 to 15_{10} now represents signed values from –8 to 7. The range about zero is asymmetrical because of the sign-bit and the fact that there is no negative 0. Once the two's complement has been obtained, subtraction is performed by adding the two's complement subtrahend to the minuend. For example, 7 – 5 = 2 would be performed as follows, given the –5 representation obtained above:

1	*1*	*1*	*1*	*0*	*Carry bits*
	0	1	1	1	Minuend (7)
+	1	0	1	1	"Subtrahend" (–5)
	0	0	1	0	Result (2)

Note that the final carry-bit past the sign-bit is ignored. An example of subtraction with a negative result is 3 – 5 = –2.

		1	*1*	*0*	*Carry bits*
	0	0	1	1	Minuend (3)
+	1	0	1	1	"Subtrahend" (–5)
	1	1	1	0	Result (–2)

Here, the result has its sign-bit set, indicating a negative quantity. We can check the answer by calculating the two's complement of the negative quantity.

This check succeeds and shows that two's complement conversions work "both ways," going back and forth between negative and positive numbers. The exception to this rule is the asymmetrical case in which the largest negative number is one more than the largest positive number as a result of the presence of the sign-bit. A four-bit number, therefore, has no positive counterpart of –8. Similarly, an 8-bit number has no positive counterpart of –128.

1.7 MULTIPLICATION AND DIVISION

Multiplication and division follow the same mathematical rules used in decimal numbering. How-

Walking through these partial products takes extra logic and time, which is why multiplication and, by extension, division are considered advanced operations that are not nearly as common as addition and subtraction. Methods of implementing these functions require trade-offs between logic complexity and the time required to calculate a final result.

1.8 FLIP-FLOPS AND LATCHES

Logic alone does not a system make. Boolean equations provide the means to transform a set of inputs into deterministic results. However, these equations have no ability to store the results of previous calculations upon which new calculations can be made. The preceding adder logic continually recalculates the sum of two inputs. If either input is removed from the circuit, the sum disappears as well. A series of numbers that arrive one at a time cannot be summed, because the adder has no means of storing a running total. Digital systems operate by maintaining *state* to advance through sequential steps in an algorithm. State is the system's ability to keep a record of its progress in a particular sequence of operations. A system's state can be as simple as a counter or an accumulated sum.

State-full logic elements called *flip-flops* are able to indefinitely hold a specific state (0 or 1) until a new state is explicitly loaded into them. Flip-flops load a new state when triggered by the transition of an input *clock*. A clock is a repetitive binary signal with a defined period that is composed of 0 and 1 phases as shown in Fig. 1.10. In addition to a defined period, a clock also has a certain *duty cycle*, the ratio of the duration of its 0 and 1 phases to the overall period. An ideal clock has a 50/50 duty cycle, indicating that its period is divided evenly between the two states. Clocks regulate the operation of a digital system by allowing time for new results to be calculated by logic gates and then capturing the results in flip-flops.

There are several types of flip-flops, but the most common type in use today is the *D flip-flop*. Other types of flip-flops include RS and JK, but this discussion is restricted to D flip-flops because of their standardized usage. A D flip-flop is often called a *flop* for short, and this terminology is used throughout the book. A basic rising-edge triggered flop has two inputs and one output as shown in Fig. 1.11a. By convention, the input to a flop is labeled D, the output is labeled Q, and the clock is represented graphically by a triangle. When the clock transitions from 0 to 1, the state at the D input is propagated to the Q output and stored until the next rising edge. State-full logic is often described through the use of a timing diagram, a drawing of logic state versus time. Figure 1.11b shows a basic flop timing diagram in which the clock's rising edge triggers a change in the flop's state. Prior to the rising edge, the flop has its initial state, Q_0, and an arbitrary 0 or 1 input is applied as D_0. The rising edge loads D_0 into the flop, which is reflected at the output. Once triggered, the flop's input can change without affecting the output until the next rising edge. Therefore, the input is labeled as "don't care," or "xxx" following the clock's rising edge.

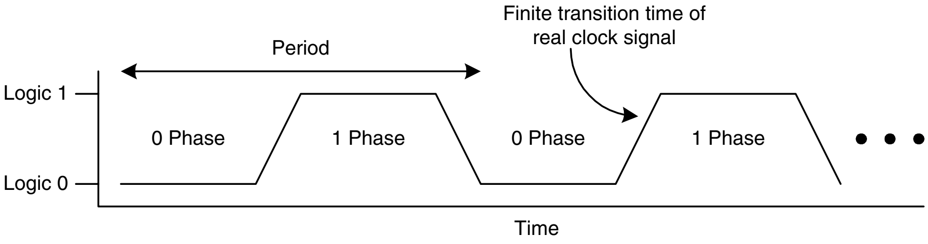
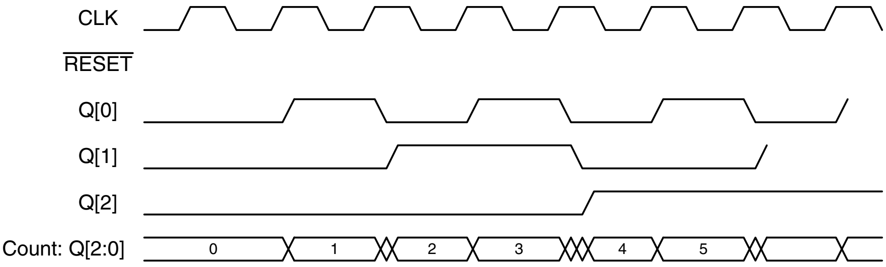


FIGURE 1.10 Digital clock signal.

Rising-edge flops are the norm, although some flops are falling-edge triggered. A falling-edge triggered flop is indicated by placing an inversion bubble at the clock input as shown in Fig. 1.12. Operation is the same, with the exception that the polarity of the clock is inverted. The remainder of this discussion assumes rising-edge triggered flops unless explicitly stated otherwise.

There are several common feature enhancements to the basic flop, including clock-enable, set, and clear inputs and a complementary output. Clock enable is used as a triggering qualifier each time a rising clock edge is detected. The D input is loaded only if clock enable is set to its active



A relative of the flop is the D-type *latch*, which is also capable of retaining its state indefinitely. A latch has a D input, a Q output, and an enable (EN) signal. Whereas a flop transfers its input to its output only on the active clock edge, a latch continuously transfers D to Q while EN is active. Latches are level sensitive, whereas flops are edge sensitive. A latch retains its state while EN is inactive. Table 1.11 shows the latch's truth table. Latches are simpler than flops and are unsuited to many applications in which flops are used. Latches would not substitute for flops in the preceding ripple counter example because, while the enable input is high, a continuous loop would be formed between the complementary output and input. This would result in rapid, uncontrolled oscillation at each latch during the time that the enable is held high.

TABLE 1.11 D-Latch Truth Table

EN	D	Q
0	X	Q_0
1	0	0
1	1	1

Latches are available as discrete logic elements and can also be assembled from simpler logic gates. The Boolean equation for a latch requires feeding back the output as follows:

When EN is high, D is passed to Q. Q then feeds back to the second AND function, which maintains the state when EN is low. Latches are used in designs based on older technology that was conceived when the latch's simplicity yielded a cost savings or performance advantage. Most state-full elements today are flops unless there is a specific benefit to using a latch.

1.9 SYNCHRONOUS LOGIC

It has been shown that clock signals regulate the operation of a state-full digital system by causing new values to be loaded into flops on each active clock edge. *Synchronous logic* is the general term

Three Boolean equations are necessary, one for each bit that feeds back to the count state flops. If the flop inputs are labeled D[2:0], the outputs are labeled Q[2:0], and an active-high synchronous reset is defined, the following equations can be developed:

$$\mathrm{D[0]} = \overline{\mathrm{Q[0]}}\&\overline{\mathrm{RESET}}$$

$$\mathrm{D[1]} = \{(\overline{\mathrm{Q[0]}}\&\mathrm{Q[1]}) + (\mathrm{Q[0]}\&\overline{\mathrm{Q[1]}})\}\&\overline{\mathrm{RESET}} = (\mathrm{Q[0]} \oplus \mathrm{Q[1]})\&\overline{\mathrm{RESET}}$$

$$+ \qquad \mathrm{D[2]}+ = \{(\overline{\mathrm{Q[2]}}\&\mathrm{Q[1]}\&\mathrm{Q[0]}) + (\mathrm{Q[2]}\&\overline{\mathrm{Q[1]}}) + (\mathrm{Q[2]}\&\overline{\mathrm{Q[0]}})\}\&\overline{\mathrm{RESET}}$$

Each equation's output is forced to 0 when RESET is asserted. Otherwise, the counter increments on each rising clock edge. Synchronous logic design allows any function to be implemented by changing the feedback logic. It would not be difficult to change the counter logic to count only odd or even numbers, or to count only as high as 5 before rolling over to 0. Unlike the ripple counter, whose structure supports a fixed counting sequence, next state logic can be defined arbitrarily according to an application's needs.

1.10 SYNCHRONOUS TIMING ANALYSIS

Logic elements, including flip-flops and gates, are physical devices that have finite response times to stimuli. Each of these elements exhibits a certain propagation delay between the time that an input is presented and the time that an output is generated. As more gates are chained together to create more complex logic functions, the overall propagation delay of signals between the end points increases. Flip-flops are triggered by the rising edge of a clock to load their new state, requiring that the input to the flip-flop is stable prior to the rising edge. Similarly, a flip-flop's output stabilizes at a new state some time after the rising edge. In between the output of a flip-flop and the input of another flip-flop is an arbitrary collection of logic gates, as seen in the preceding synchronous counter circuit. Synchronous timing analysis is the study of how the various delays in a synchronous circuit combine to limit the speed at which that circuit can operate. As might be expected, circuits with lesser delays are able to run faster.

A clock breaks time into discrete intervals that are each the duration of a single clock period. From a timing analysis perspective, each clock period is identical to the last, because each rising clock edge is a new flop triggering event. Therefore, timing analysis considers a circuit's delays over one clock period, between successive rising (or falling) clock edges. Knowing that a wide range of clock frequencies can be applied to a circuit, the question of time arises of how fast the clock can go before the circuit stops working reliably. The answer is that the clock must be slow enough to allow sufficient time for the output of a flop to stabilize, for the signal to propagate through the combinatorial logic gates, and for the input of the destination flop to stabilize. The clock must also be slow enough for the flop to reliably detect each edge. Each flop circuit is characterized by a minimum clock pulse width that must be met. Failing to meet this minimum time can result in the flop missing clock events.

Timing analysis revolves around the basic timing parameters of a flop: *input setup time* (t_{SU}), *input hold time* (t_H), and *clock-to-out time* (t_{CO}). Setup time specifies the time immediately preceding the rising edge of the clock by which the input must be stable. If the input changes too soon before the clock edge, the electrical circuitry within the flop will not have enough time to properly recognize the state of the input. Hold time places a restriction on how soon after the clock edge the input may begin to change. Again, if the input changes too soon after the clock edge, it may not be properly detected by the circuitry. Clock-to-out time specifies how soon after the clock edge the output will be updated to the state presented at the input. These parameters are very brief in duration and are usually measured in nanoseconds. One nanosecond, abbreviated "ns," is one billionth of a second. In very fast microchips, they may be measured in picoseconds, or one trillionth or a second.

Consistent terminology is necessary when conducting timing analysis. Timing is expressed in units of both clock frequency and time. Clock frequency, or speed, is quantified in units of *hertz*, named after the twentieth century German physicist, Gustav Hertz. One hertz is equivalent to one clock cycle per second—one transition from low to high and a second transition from high to low. Units of hertz are abbreviated as Hz and are commonly accompanied by prefixes that denote an order of magnitude. Commonly observed prefixes used to quantify clock frequency and their definitions are listed in Table 1.13. Unlike quantities of bytes that use binary-based units, clock frequency uses decimal-based units.

TABLE 1.13 Common Clock Frequency Magnitude Prefixes

Prefix	Definition	Order of Magnitude	Abbreviation	Usage
Kilo	Thousand	10^3	K	kHz
Mega	Million	10^6	M	MHz
Giga	Billion	10^9	G	GHz
Tera	Trillion	10^{12}	T	THz

Units of time are used to express a clock's period as well as basic logic element delays such as the aforementioned t_{SU}, t_H, and t_{CO}. As with frequency, standard prefixes are used to indicate the order of magnitude of a time specification. However, rather than expressing positive powers of ten, the exponents are negative. Table 1.14 lists the common time magnitude prefixes employed in timing analysis.

TABLE 1.14 Common Time Magnitude Prefixes

Prefix	Definition	Order of Magnitude	Abbreviation	Usage
Milli	One-thousandth	10^{-3}	m	ms
Micro	One-millionth	10^{-6}	µ	µs
Nano	One-billionth	10^{-9}	n	ns
Pico	One-trillionth	10^{-12}	p	ps

Aside from basic flop timing characteristics, timing analysis must take into consideration the finite propagation delays of logic gates and wires that connect flop outputs to flop inputs. All real components have nonzero propagation delays (the time required for an electrical signal to move from an input to an output on the same component). Wires have an approximate propagation delay of 1 ns for every 6 in of length. Logic gates can have propagation delays ranging from more than

10 ns down to the picosecond range, depending on the technology being used. Newly designed logic circuits should be analyzed for timing to ensure that the inherent propagation delays of the logic gates and interconnect wiring do not cause a flop's t_{SU} and t_H specifications to be violated at a given clock frequency.

Basic timing analysis can be illustrated with the example logic circuit shown Fig. 1.16. There are two flops connected by two gates. The logic inputs shown unconnected are ignored in this instance, because timing analysis operates on a single path at a time. In reality, other paths exist through these unconnected inputs, and each path must be individually analyzed. Each gate has a finite propagation delay, t_{PROP}, which is assumed to be 5 ns for the sake of discussion. Each flop has t_{CO} = 7 ns, t_{SU} = 3 ns, and t_H = 1 ns. For simplicity, it is assumed that there is zero delay through the wires that connect the gates and flops.

The timing analysis must cover one clock period by starting with one rising clock edge and ending with the next rising edge. How fast can the clock run? The first delay encountered is t_{CO} of the source flop. This is followed by t_{PROP} of the two logic gates. Finally, t_{SU} of the destination flop must be met. These parameters may be summed as follows:

$$t_{CLOCK} = t_{CO} + 2 \times t_{PROP} + t_{SU} = 20 \text{ ns}$$

The frequency and period of a clock are inversely related such that $F = 1/t$. A 20-ns clock period corresponds to a 50-MHz clock frequency: $1/(20 \times 10^{-9}) = 50 \times 10^6$. Running at exactly the calculated clock period leaves no room for design margin. Increasing the period by 5 ns reduces the clock to 40 MHz and provides headroom to account for propagation delay through the wires.

Hold time compliance can be verified following setup time analysis. Meeting a flop's hold time is often not a concern, especially in slower circuits as shown above. The 1 ns t_H specification is easily met, because the destination flop's D-input will not change until $t_{CO} + 2 \times t_{PROP} = 17$ ns after the rising clock edge. Actual timing parameters have variance associated with them, and the best-case t_{CO} and t_{PROP} would be somewhat smaller numbers. However, there is so much margin in this case that t_H compliance is not a concern.

Hold-time problems sometimes arise in fast circuits where t_{CO} and t_{PROP} are very small. When there are no logic gates between two flops, t_{PROP} can be nearly zero. If the minimum t_{CO} is nearly equal to the maximum t_H, the situation should be carefully investigated to ensure that the destination flop's input remains stable for a sufficient time period after the active clock edge.

1.11 CLOCK SKEW

The preceding timing analysis example is simplified for ease of presentation by assuming that the source and destination flops in a logic path are driven by the same clock signal. Although a synchronous circuit uses a common clock for all flops, there are small, nonzero variances in clock timing at individual flops. Wiring delay variances are one source of this nonideal behavior. When a clock source drives two flops, the two wires that connect to each flop's clock input are usually not identical

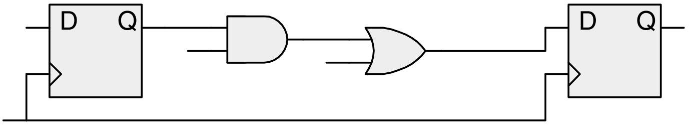


FIGURE 1.16 Hypothetical logic circuit.

in length. This length inequality causes one flop's clock to arrive slightly before or after the other flop's clock.

Clock skew is the term used to characterize differences in edge timing between multiple clock inputs. Skew caused by wiring delay variance can be effectively minimized by designing a circuit so that clock distribution wires are matched in length. A more troublesome source of clock skew arises when there are too many clock loads to be driven by a single source. Multiple clock drivers are necessary in these situations, with small variations in electrical characteristics between each driver. These driver variances result in clock skew across all the flops in a synchronous design. As might be expected, clock skew usually reduces the frequency at which a synchronous circuit can operate.

Clock skew is subtracted from the nominal clock period for setup time analysis purposes, because the worst-case scenario shown in Fig. 1.17 must be considered. This scenario uses the same logic circuit in Fig. 1.16 but shows two separate clocks with 1 ns of skew between them. The worst timing occurs when the destination flop's clock arrives before that of the source flop, thereby reducing the amount of time available for the D-input to stabilize. Instead of the circuit having zero margin with a 20-ns period, clock skew increases the minimum period to 21 ns. The extra 1 ns compensates for the clock skew to restore a minimum source to destination period time of 20 ns. A slower circuit such as this one is not very sensitive to clock skew, especially after backing off to 40 MHz for timing margin as shown previously. Digital systems that run at relatively low frequencies may not be affected by clock skew, because they often have substantial margins built into their timing analyses. As clock speeds increase, the margin decreases to the point at which clock skew and interconnect delay become important limiting factors in system design.

Hold time compliance can become more difficult in the presence of clock skew. The basic problem occurs when clock skew reduces the source flop's apparent t_{CO} from the destination flop's perspective, causing the destination's input to change before t_H is satisfied. Such problems are more prone in high-speed systems, but slower systems are not immune. Figure 1.18 shows a timing diagram for a circuit with 1 ns of clock skew where two flops are connected by a short wire with nearly zero propagation delay. The flops have t_{CO} = 2 ns and t_H = 1.5 ns. A scenario like this may be experienced when connecting two chips that are next to each other on a circuit board. In the absence of clock skew, the destination flop's input would change t_{CO} after the rising clock edge, exceeding t_H by 0.5 ns. The worst-case clock skew causes the source flop clock to arrive before that of the destination flop, resulting in an input change just 1 ns after the rising clock edge and violating t_H.

Solutions to skew-induced t_H violations include reducing the skew or increasing the delay between source and destination. Unfortunately, increasing a signal's propagation delay may cause t_{SU} violations in high-speed systems.

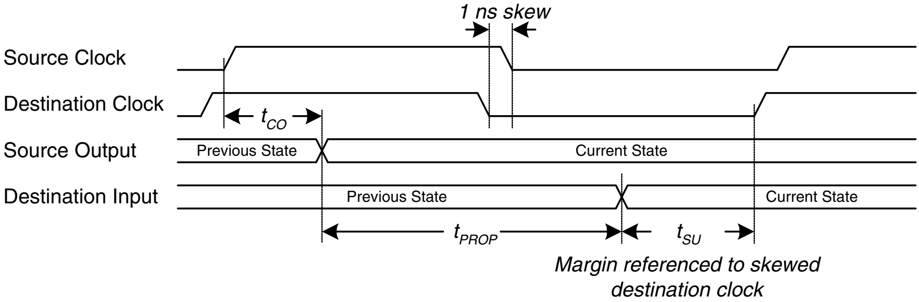


FIGURE 1.17 Clock skew influence on setup time analysis.

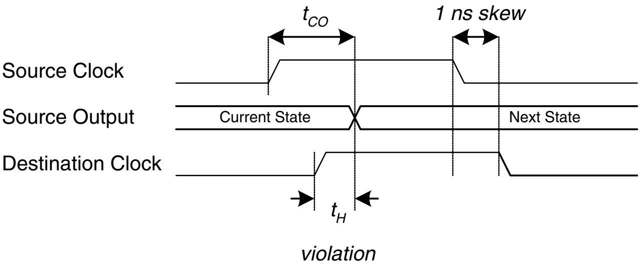


FIGURE 1.18 Hold-time violation caused by clock skew.

Hold time may not be a problem in slower circuits, because slower circuits often have paths between flops with sufficiently long propagation delays to offset clock skew problems. However, even slow circuits can experience hold-time problems if flops are connected with wires or components that have small propagation delays. It is also important to remember that hold-time compliance is not a function of clock period but of clock skew, t_{CO}, and t_H. Therefore, a slow system that uses fast components may have problems if the clock skew exceeds the difference between t_{CO} and t_H.

1.12 CLOCK JITTER

An ideal clock signal has a fixed frequency and duty cycle, resulting in its edges occurring at the exact time each cycle. Real clock signals exhibit slight variations in the timing of successive edges. This variation is known as *jitter* and is illustrated in Fig. 1.19. Jitter is caused by nonideal behavior of clock generator circuitry and results in some cycles being longer than nominal and some being shorter. The average clock frequency remains constant, but the cycle-to-cycle variance may cause timing problems.

Just as clock skew worsens the analysis for both t_{SU} and t_H, so does jitter. Jitter must be subtracted from calculated timing margins to determine a circuit's actual operating margin. Some systems are more sensitive to jitter than others. As operating frequencies increase, jitter becomes more of a problem, because it becomes a greater percentage of the clock period and flop timing specifications. Jitter specifications vary substantially. Many systems can tolerate 0.5 ns of jitter and more. Very sensitive systems may require high-quality clock circuitry that can reduce jitter to below 100 ps.

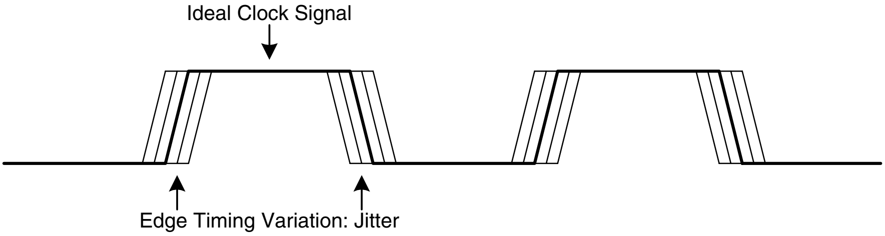


FIGURE 1.19 Clock jitter.

[redacted]

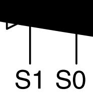


FIGURE 1.20 Four-to-one multiplexer.

FIGURE 1.21 One-to-four demultiplexer.

On each rising clock edge, a new serial input bit is clocked into the first flop, and each flop in succession loads its new value based on its predecessor's value. At any given time, the parallel output of an N-bit shift register reflects the state of the last N bits shifted in up to that time. In this example

This page intentionally left blank.

CHAPTER 2
Integrated Circuits and the 7400 Logic Families

Once basic logic design theory is understood, the next step is transferring that knowledge to a practical context that includes real components. This chapter explains what an integrated circuit is and how off-the-shelf components can be used to implement arbitrary logic functions.

Integrated circuits, called *chips* by engineers and laymen alike, are what enable digital systems as we know them. The chapter begins with an introduction to how chips are constructed. Familiarity with basic chip fabrication techniques and terminology enables an engineer to comprehend the distinctions between various products so that their capabilities can be more readily evaluated.

A survey of packaging technology follows to provide familiarity with the common physical characteristics of commercially available chips. Selecting a package that is appropriate for a particular design can be as critical as selecting the functional parameters of the chip itself. It is important to understand the variety of available chip packages and why different types of packages are used for different applications.

The chapter's major topic follows next: the 7400 logic families. These off-the-shelf logic chips have formed the basis of digital systems for decades and continue to do so, although in fewer numbers as a result of the advent of denser components. 7400 family features are presented along with complete examples of how the chips are applied in real designs. The purpose of this discussion is to impart a practical and immediately applicable understanding of how digital system design can be executed with readily available components. Although these devices are not appropriate for every application, many basic problems can be solved with 7400 chips once it is understood how to employ them.

Having seen how real chips can be used to solve actual design problems, a closely related topic is presented at the end of this chapter: the interpretation of data sheets. Manufacturers' data sheets contain critical information that must be understood to ensure a working design. An understanding of how data sheets are organized and the types of information that they contain is a necessary knowledge base for every engineer.

2.1 THE INTEGRATED CIRCUIT

Digital logic and electronic circuits derive their functionality from electronic switches called *transistors*. Roughly speaking, the transistor can be likened to an electronically controlled valve whereby energy applied to one connection of the valve enables energy to flow between two other connections. By combining multiple transistors, digital logic building blocks such as AND gates and flip-flops are formed. Transistors, in turn, are made from *semiconductors*. Consult a periodic table of elements in a college chemistry textbook, and you will locate semiconductors as a group of elements separating the metals and nonmetals. They are called semiconductors because of their ability to behave as both metals and nonmetals. A semiconductor can be made to conduct electricity like a metal or to insulate as a nonmetal does. These differing electrical properties can be accurately controlled by mixing the semiconductor with small amounts of other elements. This mixing is called *doping*. A semiconductor can be doped to contain more electrons (N-type) or fewer electrons (P-type). Examples of commonly used semiconductors are silicon and germanium. Phosphorous and boron are two elements that are used to dope N-type and P-type silicon, respectively.

A transistor is constructed by creating a sandwich of differently doped semiconductor layers. The two most common types of transistors, the *bipolar-junction transistor* (BJT) and the *field-effect transistor* (FET) are schematically illustrated in Fig. 2.1. This figure shows both the silicon structures of these elements and their graphical symbolic representation as would be seen in a circuit diagram. The BJT shown is an *NPN* transistor, because it is composed of a sandwich of N-P-N doped silicon. When a small current is injected into the *base* terminal, a larger current is enabled to flow from the *collector* to the *emitter*. The FET shown is an N-channel FET; it is composed of two N-type regions separated by a P-type substrate. When a voltage is applied to the insulated *gate* terminal, a current is enabled to flow from the *drain* to the *source*. It is called N-channel, because the gate voltage induces an N-channel within the substrate, enabling current to flow between the N-regions.

Another basic semiconductor structure shown in Fig. 2.1 is a *diode,* which is formed simply by a junction of N-type and P-type silicon. Diodes act like one-way valves by conducting current only from P to N. Special diodes can be created that emit light when a voltage is applied. Appropriately enough, these components are called *light emitting diodes*, or LEDs. These small lights are manufactured by the millions and are found in diverse applications from telephones to traffic lights.

The resulting small chip of semiconductor material on which a transistor or diode is fabricated can be encased in a small plastic package for protection against damage and contamination from the outside world. Small wires are connected within this package between the semiconductor sandwich and pins that protrude from the package to make electrical contact with other parts of the intended circuit. Once you have several discrete transistors, digital logic can be built by directly wiring these components together. The circuit will function, but any substantial amount of digital logic will be very bulky, because several transistors are required to implement each of the various types of logic gates.

At the time of the invention of the transistor in 1947 by John Bardeen, Walter Brattain, and William Shockley, the only way to assemble multiple transistors into a single circuit was to buy separate discrete transistors and wire them together. In 1959, Jack Kilby and Robert Noyce independently in-

FIGURE 2.1 BJT, FET, and diode structural and symbolic representations.

vented a means of fabricating multiple transistors on a single slab of semiconductor material. Their invention would come to be known as the *integrated circuit*, or IC, which is the foundation of our modern computerized world. An IC is so called because it integrates multiple transistors and diodes onto the same small semiconductor chip. Instead of having to solder individual wires between discrete components, an IC contains many small components that are already wired together in the desired topology to form a circuit.

A typical IC, without its plastic or ceramic package, is a square or rectangular silicon die measuring from 2 to 15 mm on an edge. Depending on the level of technology used to manufacture the IC, there may be anywhere from a dozen to tens of millions of individual transistors on this small chip. This amazing density of electronic components indicates that the transistors and the wires that connect them are extremely small in size. Dimensions on an IC are measured in units of micrometers, with one micrometer (1 µm) being one millionth of a meter. To serve as a reference point, a human hair is roughly 100 µm in diameter. Some modern ICs contain components and wires that are measured in increments as small as 0.1 µm! Each year, researchers and engineers have been finding new ways to steadily reduce these feature sizes to pack more transistors into the same silicon area, as indicated in Fig. 2.2.

Many individual chemical process steps are involved in fabricating an IC. The process begins with a thin, clean, polished semiconductor wafer — most often silicon — that is usually one of three standard diameters: 100, 200, or 300 mm. The circular wafer is cut from a cylindrical ingot of solid silicon that has a perfect crystal structure. This perfect crystal base structure is necessary to promote the formation of other crystals that will be deposited by subsequent processing steps. Many dice are arranged on the wafer in a grid as shown in Fig. 2.3. Each die is an identical copy of a master pattern and will eventually be sliced from the wafer and packaged as an IC. An IC designer determines how different portions of the silicon wafer should be modified to create transistors, diodes, resistors, capacitors, and wires. This IC design layout can then be used to, in effect, draw tiny components onto the surface of the silicon. Sequential drawing steps are able to build sandwiches of differently doped silicon and metal layers.

Engineers realized that light provided the best way to faithfully replicate patterns from a template onto a silicon substrate, similar to what photographers have been doing for years. A photographer takes a picture by briefly exposing film with the desired image and then developing this film into a negative. Once this negative has been created, many identical photographs can be reproduced by briefly exposing the light-sensitive photographic paper to light that is focused through the negative. Portions of the negative that are dark do not allow light to pass, and these corresponding regions of the paper are not exposed. Those areas of the negative that are light allow the paper to be exposed.

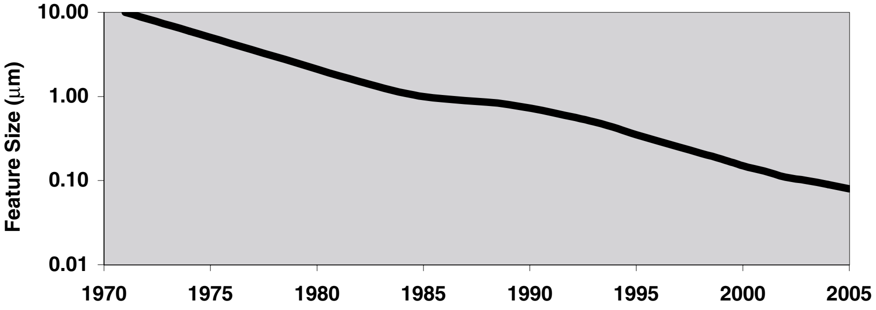


FIGURE 2.2 Decreasing IC feature size over time. *(Future data for years 2003 through 2005 compiled from* The International Technology Roadmap for Semiconductors, *Semiconductor Industry Association, 2001.)*

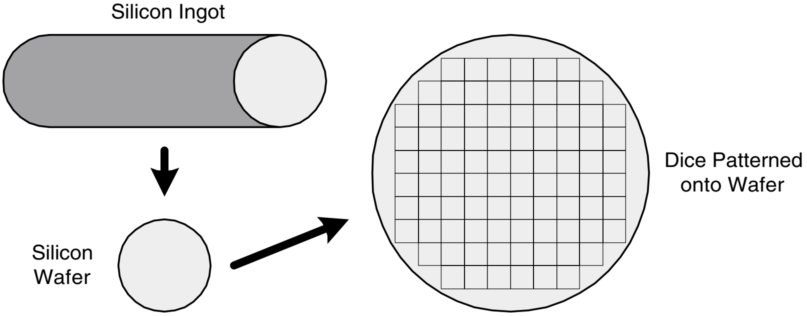


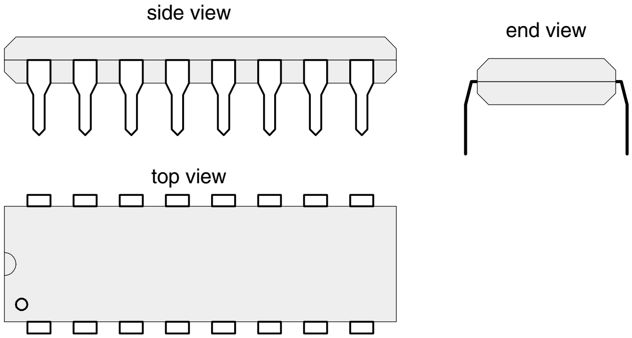
FIGURE 2.3 Silicon wafer.

When the paper is developed in a chemical bath, portions of the paper that were exposed change color and yield a visible image.

Photographic processes provide excellent resolution of detail. Engineers apply this same principle in fabricating ICs to create details that are fractions of a micron in size. Similar to a photographic negative, a mask is created for each IC processing step. Like a photographic negative, the mask does not have to be the same size as the silicon area it is to expose because, with lenses, light can be focused through the mask to an arbitrary area. Using a technique called *photolithography,* the silicon surface is first prepared with a light-sensitive chemical called *photoresist*. The prepared surface is then exposed to light through the mask. Depending on whether a positive or negative photoresist process is employed, the areas of photoresist that have been either exposed or not exposed to light are washed away in a chemical bath, resulting in a pattern of bare and covered areas of silicon. The wafer can then be exposed to chemical baths, high temperature metal vapors, and ion beams. Only the bare areas that have had photoresist washed away are affected in this step. In this way, specific areas of the silicon wafer can be doped according to the IC designers' specifications. Successive mask layers and process steps can continue to wash away and expose new layers of photoresist and then build sandwiches of semiconductor and metal material. A very simplified view of these process steps is shown in Fig. 2.4. The semiconductor fabrication process must be performed in a clean-room environment to prevent minute dust particles and other contaminants from disturbing the lithography and chemical processing steps.

In reality, dozens of such steps are necessary to fabricate an IC. The semiconductor structures that must be formed by layering different metals and dopants are complex and must be formed one thin layer at a time. Modern ICs typically have more than four layers of metal, each layer separated from others by a thin insulating layer of silicon dioxide. The use of more metal layers increases the cost of an IC, but it also increases its density, because more metal wires can be fabricated to connect more transistors. This complete process from start to finish usually takes one to four weeks. The chemical diffusion step (5) is an example of how different regions of the silicon wafer are doped to achieve varying electrical characteristics. In reality, several successive doping steps are required to create transistors. The metal deposition step (10) is an example of how the microscopic metal wires that connect the many individual transistors are created. Hot metal vapors are passed over the prepared surface of the wafer. Over time, individual molecules adhere to the exposed areas and form continuous wires. Historically, most metal interconnects on silicon ICs are made from aluminum. However, copper has become a common component of leading-edge ICs.

As IC feature sizes continue to shrink, the physical properties of light can become limiting factors in the resolution with which a wafer can be processed. Shorter light wavelengths are necessary to



metal frame and is connected to the individual pins with extremely thin wires. Once the electrical connections are made, the fragile assembly is encased in a plastic or ceramic body for protection and the exterior portions of the pins are folded vertically.

All other IC packages are variations on this theme. Some packages use a similar lead-frame structure, whereas more advanced packages utilize very high-quality miniature circuit boards made from either ceramic or fiberglass.

An oft-quoted attribute of ICs is that their density doubles every 18 months as a result of improvements in process technology. This prediction was made in 1965 by Dr. Gordon Moore, a co-founder of Intel. It has since come to be known as *Moore's law,* because the semiconductor industry has matched this prediction over time. Before to the explosion of IC density, the semiconductor industry classified ICs into several categories depending on the number of logic gates on the device: *small-scale integration* (SSI), *medium-scale integration* (MSI), *large-scale integration* (LSI), and, finally, *very large-scale integration* (VLSI). Figure 2.7 provides a rough definition of these terms. As the density of ICs continued to grow at a rapid pace, it became rather ridiculous to keep adding words like "very" and "extra" to these categories, and the terms' widespread use declined. ICs are now often categorized based on their minimum feature size and metal process. For example, one might refer to an IC as "0.25 µm, three-layer metal (aluminum)" or "0.13 µm, six-layer copper."

8-, 12-, and 13-input NAND gates. There are numerous varieties of flip-flops, counters, multiplexers, shift registers, and bus transceivers. Flip-flops exist with and without complementary outputs, preset/clear inputs, and independent clocks. Counters are available in 4-bit blocks that can both increment and decrement and count to either 15 (binary counter) or 9 (decade counter) before restarting the count at 0. Shift registers exist in all permutations of serial and parallel inputs and outputs. Bus transceivers in 4- and 8-bit increments exist with different types of output enables and capabilities to function in unidirectional or bidirectional modes. Bus transceivers enable the creation and expansion of tri-state buses on which multiple devices can communicate.

One interesting IC is the 7447 seven-segment display driver. This component allows the creation of graphical numeric displays in applications such as counters and timers. Seven-segment displays are commonly seen in automobiles, microwave ovens, watches, and consumer electronics. Seven independent on/off elements can represent all ten digits as shown in Fig. 2.11. The 7447 is able to drive an LED-based seven-segment display when given a *binary coded decimal* (BCD) input. BCD is a four-bit binary number that has valid values from 0 through 9. Hexadecimal values from 0xA through 0xF are not considered legal BCD values.

Familiarity with the 7400 series proves very useful no matter what type of digital system you are designing. For low-end systems, 7400-series logic may be the only type of IC at your disposal to solve a wide range of problems. At the high end, many people are often surprised to see a small 14-pin 7400-series IC soldered to a circuit board alongside a fancy 32-bit microprocessor running at 100 MHz. The fact is that the basic logic functions that the 7400 series offers are staples that have direct applications at all levels of digital systems design. It is time well spent to become familiar with the extensive capabilities of the simple yet powerful 7400 family. Manufacturers' logic data books, either in print or on line, are invaluable references. It can be difficult to know ahead of time if a design may call for one more gate to function properly; that is when a 40-year old logic family can save the day.

2.4 APPLYING THE 7400 FAMILY TO LOGIC DESIGN

Applications of the 7400 family are truly infinite, because the various ICs represent basic building blocks rather than complete solutions. Up through the early 1980s, it was common to see computer systems constructed mainly from interconnected 7400-series ICs along with a few LSI components such as a microprocessor and a few memory chips. These days, most commercial digital systems are designed using some form of higher-density logic IC, either fully custom or user programmable. However, the engineer or hobbyist who has a relatively small-scale logic problem to solve, and who may not have access to more expensive custom or programmable logic ICs, may be able to utilize only 7400 logic in an efficient and cost-effective solution. Two examples follow to provide insight into how 7400 building blocks can be assembled to solve logic design problems.

A hypothetical example is a logic circuit to examine three switches and turn on an LED if two and only two of the three switches are turned on. The truth table for such a circuit is as follows in

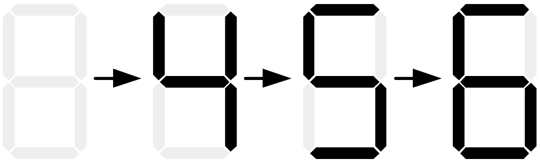


Table 2.2, given that A, B, and C are the inputs, and an LED is the active-low output (assume that the LED is turned on by driving a logic 0 rather than a logic 1).

TABLE 2.2 LED Driver Logic Truth Table

A	B	C	$\overline{\text{LED}}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

This LED driver truth table can be converted into the following Boolean logic equation with a Karnaugh map or simply by inspection:

$$\overline{\text{LED}} = \overline{\bar{A}BC + A\bar{B}C + AB\bar{C}}$$

After consulting a list of available 7400 logic ICs, three become attractive for our application: the 7404 inverter, 7408 AND, and 7432 OR. The LED driver logic equation requires four inverters, six two-input AND gates, and two 2-input OR gates. Four ICs are required, because a 7404 provides six inverters, a 7408 provides four AND gates, and a 7432 contains four OR gates. These four ICs can be connected according to a *schematic diagram* as shown in Fig. 2.12. A schematic diagram illustrates the electrical connectivity scheme of various components. Each component is identified by a *reference designator* consisting of a letter followed by a number. ICs are commonly identified by reference designators beginning with the letter "U". Additionally, each component has numerous pins that are numbered on the diagram. These pin numbers conform to the IC manufacturer's numbering scheme. Each of these 7400-series ICs has 14 pins. Another convention that remains from bipolar logic days is the use of the label VCC to indicate the positive voltage supply node. GND represents ground—the common, or return, voltage supply node.

All ICs require connections to a power source. In this circuit, +5 V serves as the power supply, because the 7400 family is commonly manufactured in a bipolar semiconductor process requiring a +5-V supply. The four rectangular blocks at the top of the diagram represent this power connection information. Because this schematic diagram shows individual gates, the gates' reference designators contain an alphabetic suffix to identify unique instances of gates within the same IC. Not all gates in each IC are actually used. Those that are unused are tied inactive by connecting their inputs to a valid logic level—in this case, ground. It would be equally valid to connect the inputs of unused gates to the positive supply voltage, +5 V.

This logic circuit would work, but a more efficient solution is available to those who are familiar with the capabilities of the 7400 family. The 7411 provides three 3-input AND gates, which is perfect for this application, allowing a reduction in the part count to three ICs instead of four. This cir-

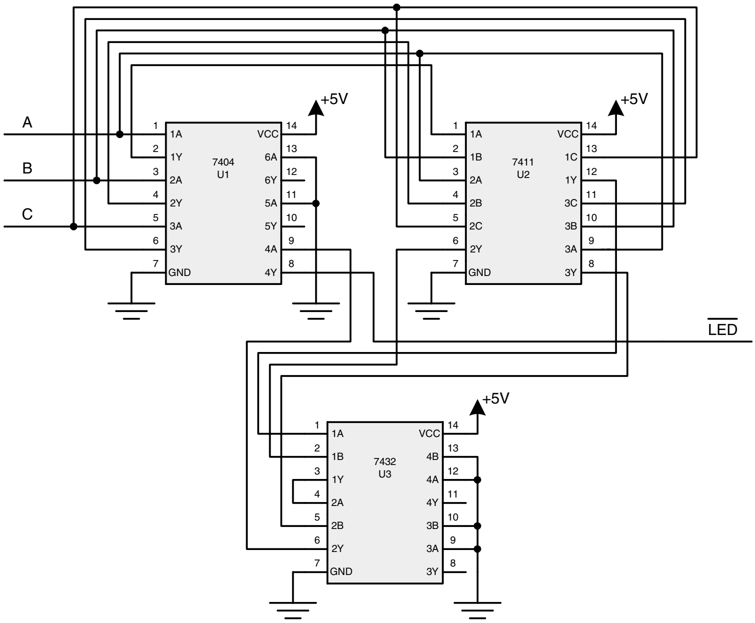


FIGURE 2.13 LED driver logic using 74111 with fewer ICs.

detected, the last bit is found by knowing that there are eight bits in a byte. During periods of inactivity, an idle communications interface is indicated by a persistent logic 0. When the transmitter is given a byte to send, it first drives a logic-1 start bit and then sends eight data bits. Each bit is sent in its own clock cycle. Therefore, nine clock cycles are required to transfer each byte. The serial interface is composed of two signals, *clock* and *serial data*, and functions as shown in Fig. 2.14.

The eight data bits are sent from least-significant bit, bit 0, to most-significant bit, bit 7, following the start bit. Following the transmission of bit 7, it is possible to immediately begin a new byte by inserting a new start bit. This timing diagram does not show a new start bit directly following bit 7. The corresponding output of the receiver is shown in Fig. 2.15. Here, *data out* is the eight-bit quan-

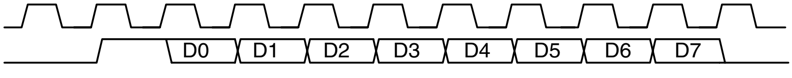


FIGURE 2.14 Serial interface bit timing.

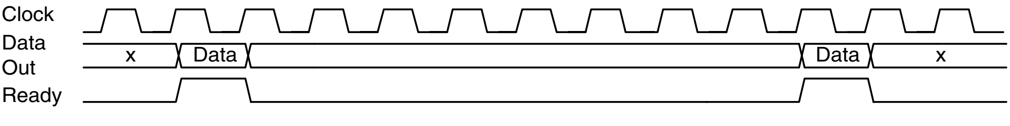


FIGURE 2.15 Serial receive output timing.

tity that has been reconstructed from the serialized bit stream of Fig. 2.14. *Ready* indicates when data out is valid and is active-high.

All that is required of this receiver is to assemble the eight data bits in their proper order and then generate a *ready* signal. This ready signal lasts only one cycle, and any downstream logic waiting for the newly arrived byte must process it immediately. In a real system, a register might exist to capture the received byte when ready goes active. This register would then pass the byte to the appropriate destination. This output timing shows two bytes transmitted back to back. They are separated by nine cycles, because each byte requires an additional start bit for framing.

In contemplating the design of the receive portion of the serial controller, the need for a serial-in/parallel-out shift register becomes apparent to assemble the individual bits into a whole byte. Additionally, some control logic is necessary to recognize the start bit, wait eight clocks to assemble the incoming byte, and then generate a ready signal. This receiver has two basic states, or modes, of operation: idle and receiving. When idling, no start bit has yet been detected, so there is no useful work to be done. When receiving, a start bit has been observed, incoming bits are shifted into the shift register, and then a ready signal is generated. As soon as the ready signal is generated, the receiver state may return to idle or remain in receiving if a new start bit is detected. Because there are two basic control logic states, the state can be stored in a single flip-flop, forming a two-state *finite state machine* (FSM). An FSM is formed by one or more *state flops* with accompanying logic to generate a new state for the next clock cycle based on the current cycle's state. The state is represented by the combined value of the state flops. An FSM with two state flops can represent four unique states. Each state can represent a particular step in an algorithm. The accompanying *state logic* controls the FSM by determining when it is time to transition to a new piece of the algorithm—a new state.

In the serial receive state machine, transitioning from idle to receiving can be done according to the serial data input, which is 0 when inactive and 1 when indicating a start bit. Transitioning back to idle must somehow be done nine cycles later. A counter could be used but would require some logic to sense a particular count value. Instead, a second shift register can be used to delay the start bit by nine cycles. When the start bit emerges from the last output bit in the shift register, the state machine can return to the idle state. Consider the logic in Fig. 2.16. The arrow-shaped boxes indicate connection points, or ports, of the circuit.

Under an idle condition, the input to the shift register is zero until the start bit appears at the data input, *din*. Nine cycles later, the ready bit emerges from the shift register. As soon as the start bit is observed, the state machine transitions to the receiving state, changing the *idle* input to 0, effectively masking further input to the shift register. This masking prevents nonzero data bits from entering the *ready* delay logic and causing false results.

Delaying the start bit by nine cycles solves one problem but creates another. The transition of the state machine back to idle is triggered by the emergence of *ready* from the shift register. Therefore, this transition will actually occur *ten* cycles after the start bit, because the state flop, like all D flip-flops, requires a single cycle of latency to propagate its input to its output. This additional cycle will prevent the control logic from detecting a new start bit immediately following the last data bit of the byte currently in progress. A solution is to design *ready* machine can look ahead one cycle into the future and return to idle in time for a new start bit that may be arriving. With the logical details of the state machine now complete, the state machine can be represented with the *state transition diagram* in Fig. 2.17.

A state transition diagram, often called a *bubble diagram*, shows all the states of an FSM and the logical *arcs* that dictate how one state leads to another. When implemented, the arcs are translated into the state logic to make the FSM function. With a clearly defined state transition diagram, the logic to drive the state machine can be organized as shown in Table 2.3.

When in the idle state (1), a high on *din* (the start bit) must be observed to transition to the receiving state (0). Once in the receiving state, *ready_next* must be high to return to idle. This logic is represented by the Boolean equation,

As with most problems, there exists more than one solution. Depending on the components available, one may choose to design the logic differently to make more efficient use of those components. As a general rule, it is desirable to limit the number of ICs used. The 7451 provides two "AND-OR-INVERT" gates, each of which implements the Boolean function,

This function is tantalizingly close to what is required for the state machine. It differs in that the inversion of two inputs (*state* and *din*) and a NOR function rather than an OR are necessary. Both differences can be resolved using a 7404 inverter IC, but there is a more efficient solution using the

In this logic circuit, the inverted output of the state flop, U1B, is used as the *state* bit to compensate for the 7451's NOR function. The unused *clr_* and *b* pins of U3 are connected to +5 V to render them neutral on the shift register's behavior. The shift register will not clear itself, because *clr_* is active-low and, similarly, the internal input AND-gate that combines *a* and *b*, will be logically bypassed by tying *b* to logic 1. The parallel byte output of this serial receiver is designated *Dout[7:0]* and is formed by grouping the eight outputs of the shift register into a single bus. One common notation for assigning members of a bus is to connect each individual member to a thicker line with some type of *bus-ripper* line. The bus ripper is often drawn in the schematic diagram as mitered or curved at the bus end to make its function more visually apparent.

Designing an accompanying serial transmitter follows a very similar design process to the preceding discussion. It is left as an exercise to the reader.

2.6 COMMON VARIANTS OF THE 7400 FAMILY

In the 1970s and 1980s, the 7400 family was commonly manufactured in a bipolar semiconductor process that operated using a +5-V power supply and was known as transistor-transistor logic (TTL). The discussion of the 7400 family thus far has included only the original +5-V bipolar type. The 7400's popularity and broad application to digital design has kept it relevant through many improvements in semiconductor process technology. As engineers learned to fabricate faster and more efficient ICs, the 7400 was redesigned in many different process generations beginning in the late 1960s. Some of the more common 7400 variants are briefly discussed here.

The original 7400 discrete TTL logic family featured typical propagation delays of 10 ns per gate and power consumption, also called *power dissipation*, of approximately 10 mW per gate. By modern standards, the 7400's speed is relatively slow, and its power dissipation is relatively high. Increasing system complexity dictates deeper logic: more gates chained together to implement more complex Boolean functions. Each added level of logic adds at least another gate's worth of propagation delay. At the same time, power consumption also becomes a problem. Ten milliwatts may not sound like a lot of power, but, when multiplied by several thousand gates, it represents a substantial design problem in terms of both supplying a large quantity of power and cooling the radiated heat from digital systems.

Two notable bipolar variants of the 7400 are the 74LS and 74F families. The 74LS, LS indicating *low-power Schottky*, has speed comparable to that of the original 7400, but it dissipates roughly 20 percent of its power. The 74F, F indicating fast, is approximately 80 percent faster than the 7400 and reduces power consumption by almost half. Whether the concern is reducing power or increasing speed, these two families are useful for applications requiring 5-V bipolar technology.

CMOS technology began to emerge in the 1980s as a popular process for fabricating digital ICs as a result of its lower power consumption as compared to bipolar. The low-power characteristics of CMOS logic stem from the fact that a FET requires essentially no current to keep it in an on or off state (unlike a BJT, which always draws some current when it is turned on). A CMOS gate, therefore, will draw current only when it switches. For this reason, the power consumption of a CMOS logic gate is extremely low in an idle, or quiescent, state and increases with the frequency at which it switches.

Several CMOS 7400 families were introduced, among them being the 74HCT and 74ACT, each of which has power consumption orders of magnitude less than bipolar equivalents at low frequencies. Earlier CMOS versions of the 7400 were not fully compatible with the bipolar devices, because of voltage threshold differences between the CMOS and bipolar processes. A typical TTL output is only guaranteed to rise above 2.5 V, depending on output loading. In contrast, a typical 5-V CMOS input requires a minimummw D©G8—A TG—-V5 haT8—min finp-mmw D©Ape x8—haT8p reD—7cpe haT
a U[*fiw$©Aa}*fio*SoUo÷©Gcnyfvic o©8©8Ufa}*fio*SG*fio*SG*fiBpHaC xfiw$©A

voltage range causes a fundamental problem in which a TTL gate driving an ordinary CMOS gate cannot be guaranteed to operate in all situations. Both the 74HCT and 74ACT families possess the low-power benefits of CMOS technology and retain compatibility with bipolar ICs. A 74HCT device is somewhat slower than a 74LS equivalent, and the 74ACT is faster than a 74LS device.

There has been an explosion of 7400 variants. Most of the families introduced in the last decade are based on CMOS technology and are tailored to a broad set of applications ranging from simple speed to high-power bus drivers. Most types of 7400 devices share common pin-outs and functions, with the exception of some proprietary specialized parts that may be produced by only a single manufacturer. Most of the 7400 families still require +5-V supplies, but lower voltages such as 3.3 V, 2.5 V, 1.8 V, and 1.5 V are available as well. These lower-voltage families are important because of the general trend toward lower voltages for digital logic.

2.7 INTERPRETING A DIGITAL IC DATA SHEET

Semiconductor manufacturers publish data sheets for each of their products. Regardless of the specific family or device, all logic IC data sheets share common types of information. Once the basic data sheet terminology and organization is understood, it is relatively easy to figure out other data sheets even when their exact terminology changes. Data sheet structure is illustrated using the 74LS00 from Fairchild Semiconductor as an example. A page from its data sheet is shown in Fig. 2.19.

Digital IC data sheets should have at least two major sections: functional description and electrical specifications. The functional description usually contains the device pin assignment, or *pin-out*, as well as a detailed discussion of how the part logically operates. A simple IC such as the 74LS00 will have a very brief functional description, because there is not much to say about a NAND gate's operation. More complex ICs such as microprocessors can have functional descriptions that fill dozens or hundreds of pages and are broken into many chapters. Some data sheets add additional sections to present the mechanical dimensions of the package and its thermal properties. Digital IC electrical specifications are similar across most types of devices and often appear in the following four categories:

- *Absolute maximum ratings.* As the term implies, these parameters specify the absolute extremes that the IC may be subjected to without sustaining permanent damage. Manufacturers almost universally state that the IC should never be operated under these extreme conditions. These ratings are useful, because they indicate how the device may be stored and express the quality of design
ture of the physical chip. Manufacturers specify a storage temperature range within
uctor structures will not break down. In the case of Fairchild's 74LS00, this
ximum voltage levels are also specified, 7 V in the case of the 74LS00,
be subjected to a 7-V potential without destructing.

s. These parameters specify the normal range of voltages and
erated within such that its functionality is guaranteed to meet
turer. Two of the most important specifications in this sec-
beled as either V_{CC} or V_{DD}, depending on whether a bipo-
temperature. An IC may have multiple supply voltage
operate on several different voltages simultaneously.
portion of the chip. When the manufacturer specifies
ually either ±5 or ±10 percent. Many 5-V
75 to 5.25 V (±5 percent). Op-
As a semiconduc-

tor heats up, it slows down. As it cools, its speed increases. Outside of the recommended operating temperature, the device is not guaranteed to function, because the effects of temperature become so severe that functionality is compromised. There are four common temperature ranges for ICs: commercial (0 to 70°C), industrial (–40 to 85°C), automotive (–40 to 125°C), and military (–55 to 125°C). It is more difficult to manufacture an IC that operates over wider temperature ranges. As such, more demanding temperature grades are often more expensive than the commercial grade.

Other parameters establish the safe operating limits for input signals as well as the applied voltage thresholds that represent logic 0 and 1 states. Minimum and maximum input levels are expressed as either absolute voltages or voltages relative to the supply voltage pins of the device. Exceeding these voltages may damage the device. Logic threshold specifications are provided to ensure that the logic input voltages are such that the device will function as intended and not confuse a 1 for a 0, or vice versa. There is also a limit to how must current a digital output can drive. Current output specifications should be known so that a chip is not overloaded, which could result in either permanent damage to the chip or the chip's failure to meet its published specifications.

- *DC electrical characteristics.* DC parameters specify the voltages and currents that the IC will present to other circuitry to which it is connected. Whereas recommended operating conditions specify the environment under which the chip will properly operate, DC electrical characteristics

Timing specifications may also be incomplete. Manufacturers do not always guarantee minimum or maximum parameters, depending on the specific type of device and the particular specification. As with DC voltages, worst-case parameters should always be specified. When a minimum or maximum delay is not specified, it is generally because that parameter is of secondary importance, and the manufacturer was unable to control its process to a sufficient level of detail to guarantee that value. In many situations where incomplete specifications are given, there are acceptable reasons for doing so, and the lack of information does not hurt the quality of the design.

Typical timing numbers are not useful in many circumstances, because they do not represent a limit of the device's operation. A thorough design must take into account the best and worst performance of each IC in the circuit so that one can guarantee that the circuit will function under all conditions. Therefore, worst-case timing parameters are usually the most important to consider first, because they are the dominant limit of a digital system's performance in most cases. In more advanced digital systems, minimum parameters can become equally as important because of the need to meet hold time and thereby ensure that a signal does not disappear too quickly before the driven IC can properly sense the signal's logic level.

Output timing specifications are often specified with an assumed set of loading conditions, because the current drawn by the load has an impact on the output driver's ability to establish a valid logic level. A small load will enable the IC to switch its output faster, because less current is demanded of the output. A heavier load has the opposite effect, because it draws more current, which places a greater strain on the output driver.

CHAPTER 3
Basic Computer Architecture

Microprocessors are central components of almost all digital systems, because combinations of hardware and software are used to solve design problems. A computer is formed by combining a microprocessor with a mix of certain basic elements and customized logic. Software runs on a microprocessor and provides a flexible framework that orchestrates the behavior of hardware that has been customized to fit the application. When many people think about computers, images of desktop PCs and laptops come to their minds. Computers are much more diverse than the stereotypical image and permeate everyday life in increasing numbers. Small computers control microwave ovens, telephones, and CD players.

Computer architecture is fundamental to the design of digital systems. Understanding how a basic computer is designed enables a digital system to take shape by using a microprocessor as a central control element. The microprocessor becomes a programmable platform upon which the major components of an algorithm can be implemented. Digital logic can then be designed to surround the microprocessor and assist the software in carrying out a specific set of tasks.

The first portion of this chapter explains the basic elements of a computer, including the microprocessor, memory, and input/output devices. Basic microprocessor operation is presented from a hardware perspective to show how instructions are executed and how interaction with other system components is handled. Interrupts, registers, and stacks are introduced as well to provide an overall picture of how computers function. Following this basic introduction is a complete example of how an actual eight-bit computer might be designed, with detailed descriptions of bus operation and address decoding.

Once basic computer architecture has been discussed, common techniques for improving and augmenting microprocessor capabilities are covered, including direct memory access and bus expansion. These techniques are not relegated to high-end computing but are found in many smaller digital systems in which it is more economical to add a little extra hardware to achieve feature and performance goals instead of having to use a microprocessor that may be too complex and more expensive than desired.

The chapter closes with an introduction to assembly language and microprocessor addressing modes. Writing software is not a primary topic of this book, but basic software design is an inseparable part of digital systems design. Without software, a computer performs no useful function. Assembly language basics are presented in a general manner, because each microprocessor has its own instruction set and assembly language, requiring specific reading focused on that particular device. Basic concepts, however, are universal across different microprocessor implementations and serve to further explain how microprocessors actually function.

execute a program that is located on the disk drive, that program is first loaded into the computer's RAM and then executed from a region set aside for program memory. As on a desktop computer, RAM is most often *volatile*—meaning that it loses its contents when the power is turned off.

Some software cannot be stored in volatile memory, because basic initialization instructions, or *boot code*, must be present when the computer is turned on. Remember that a microprocessor can do nothing useful without software being readily available. When power is first applied to a computer, the microprocessor must be able to quickly locate boot code so that it can get itself ready to accept input from a user or load a program from an input device. This startup sequence is called *booting*, hen …9startuHfe8©©eadyfAhPax[*fio88He9©xnR9—±of,©Gcx*'©A©Go9GODUwOEOC9HHyfAhPaU

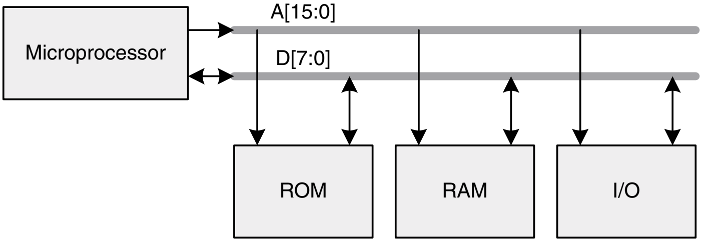


FIGURE 3.3 Microprocessor buses.

A microprocessor's entire address space is never occupied by a single function; rather, it is shared by ROM, RAM, and various I/Os. Each device is *mapped* into its own region of the address space and is enabled only when the microprocessor asserts an address within a device's mapped region. The process of recognizing that an address is within a desired region is called *decoding*. Address decoding logic is used to divide the overall address space into smaller sections in which memory and I/O devices can reside. This logic generates individual signals that enable the appropriate device based on the state of the address bus so that the devices themselves do not need any knowledge of the specific computer's unique address decoding.

3.2 MICROPROCESSOR INTERNALS

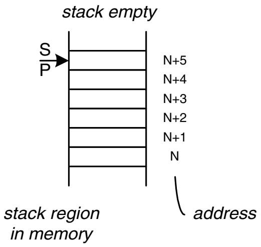
The multitude of complex tasks performed by computers can be broken down into sequences of simple operations that manipulate individual numbers and then make decisions based on those calculations. Certain types of basic instructions are common across nearly every microprocessor in existence and can be classified as follows for purposes of discussion:

- Arithmetic: add or subtract two values
- Logical: Boolean (e.g., AND, OR, XOR, NOT, etc.) manipulation of one or two values
- Transfer: retrieve a value from memory or store a value to memory
- Branch: jump ahead or back to a particular instruction if a specified condition is satisfied

Arithmetic and logical instructions enable the microprocessor to modify and manipulate specific pieces of data. Transfer instructions enable these data to be saved for later use and recalled when necessary from memory. Branch operations enable instructions to execute in different sequences, depending on the results of arithmetic and logical operations. For example, a microprocessor can compare two numbers and take one of two different actions if the numbers are equal or unequal.

Each unique instruction is represented as a binary value called an *opcode*. A microprocessor fetches and executes opcodes one at a time from program memory. Figure 3.4 shows a hypothetical microprocessor to serve as an example for discussing how a microprocessor actually advances through and executes the opcodes that form programs.

A microprocessor is a synchronous logic element that advances through opcodes on each clock cycle. Some opcodes may be simple enough to execute in a single clock cycle, and others may take multiple cycles to complete. Clock speed is often used as an indicator of a microprocessor's performance. It is a valid indicator but certainly not the only one, because each microprocessor requires a different number of cycles for each instruction, and each instruction represents a different quantity of useful work.



The stack can store multiple entries, enabling multiple subroutines to be active at the same time. If one subroutine calls another, the microprocessor must keep track of both subroutines' return addresses in the order in which the subroutines have been called. This subroutine *nesting* process of one calling another subroutine, which calls another subroutine, naturally conforms to the last-in, first-out operation of a stack.

To implement a stack, a microprocessor contains a stack pointer register that is loaded by the programmer to establish the initial starting point, or top, of the stack. Figure 3.6 shows the hypothetical microprocessor in more complete form with a stack pointer register.

Like the PC, the SP is a counter that is automatically modified by certain instructions. Not only do subroutine branch and return instructions use the stack, there are also general-purpose push/pop instructions provided to enable the programmer to use the stack manually. The stack can make certain calculations easier by pushing the partial results of individual calculations and then popping them as they are combined into a final result.

The programmer must carefully manage the location and size of the stack. A microprocessor will freely execute subroutine call, subroutine return, push, and pop instructions whenever they are encountered in the software. If an empty stack is popped, the microprocessor will oblige by reading back whatever data value is present in memory at the time and then incrementing the SP. If a full stack is pushed, the microprocessor will write the specified data to the location pointed to by the SP and then decrement it. Depending on the exact circumstances, either of these operations can corrupt other parts of the program or data that happens to be in the memory location that gets overwritten. It is the programmer's responsibility to leave enough free memory for the desired stack depth and then to not nest too many subroutines simultaneously. The programmer must also ensure that there is symmetry between push/pop and subroutine call/return operations. Issuing a return-from-subroutine instruction while already in the main program would lead to undesirable results when the micropro-

Multiple interrupt sources are common in microprocessors. Depending on the complexity of the microprocessor, there may be one, two, ten, or dozens of separate interrupt sources, each with its own vector. Conflicts in which multiple interrupt sources are activated at the same time are handled by assigning priorities to each interrupt. Interrupt priorities may be predetermined by the designer of the microprocessor or programmed by software. In a microprocessor with multiple interrupt priorities, once a higher-priority interrupt has taken control and its ISR is executing, lower-priority interrupts will remain pending until the current higher-priority ISR issues a return-from-interrupt.

Interrupts can usually be turned off, or *masked*, by writing to a control register within the microprocessor. Masking an interrupt is useful, because an interrupt should not be triggered before the program has had a chance to set up the ISR or otherwise get ready to handle the interrupt condition. If the program is not yet ready and the microprocessor takes an interrupt by jumping to the interrupt vector, the microprocessor will crash by executing invalid instructions.

Masking is also useful when performing certain time-critical operations. A task may be programmed into an ISR that must complete within 10 μs. Under normal circumstances, the task is easily accomplished in this period of time. However, if a competing interrupt is triggered during the time-critical ISR, there may be no guarantee of meeting the 10-μs requirements. One solution to this problem is to mask subsequent interrupts when the time-critical interrupt is triggered and then unmask interrupts when the ISR has completed. If an interrupt arrives while masked, the microprocessor will remember the interrupt request and trigger the interrupt when it is unmasked.

Certain microprocessors have one or more interrupts that are classified as nonmaskable. This means that the interrupt cannot be disabled. Therefore, the hardware design of the computer must ensure that such an interrupt is not activated unless the software is able to respond to it. Nonmaskable interrupts are generally used for low-level error recovery or debugging purposes where it must be guaranteed that the interrupt will be taken regardless of what the microprocessor is doing at the time. Nonmaskable ISRs are sometimes implemented in nonvolatile memory to ensure that they are always ready for execution.

3.5 IMPLEMENTATION OF AN EIGHT-BIT COMPUTER

Having discussed some of the basic principles of microprocessor architecture and operation, we can examine how a microprocessor fits into a system to form a computer. Microprocessors need external memory in which to store their programs and the data upon which they operate. In this context, external memory is viewed from a logical perspective. That is, the memory is always external to the core microprocessor element. Some processor chips on the market actually contain a certain quantity of memory within them, but, logically speaking, this memory is still external to the actual microprocessor core.

In the general sense, a computer requires a quantity of nonvolatile memory, or ROM, in which to store the boot code that will be executed on reset. The ROM may contain all or some of the microprocessor's full set of software. A small embedded computer, such as the one in a microwave oven, contains all its software in ROM. A desktop computer contains very little of its software in ROM. A computer also requires a quantity of volatile memory, or RAM, that can be used to store data associ-

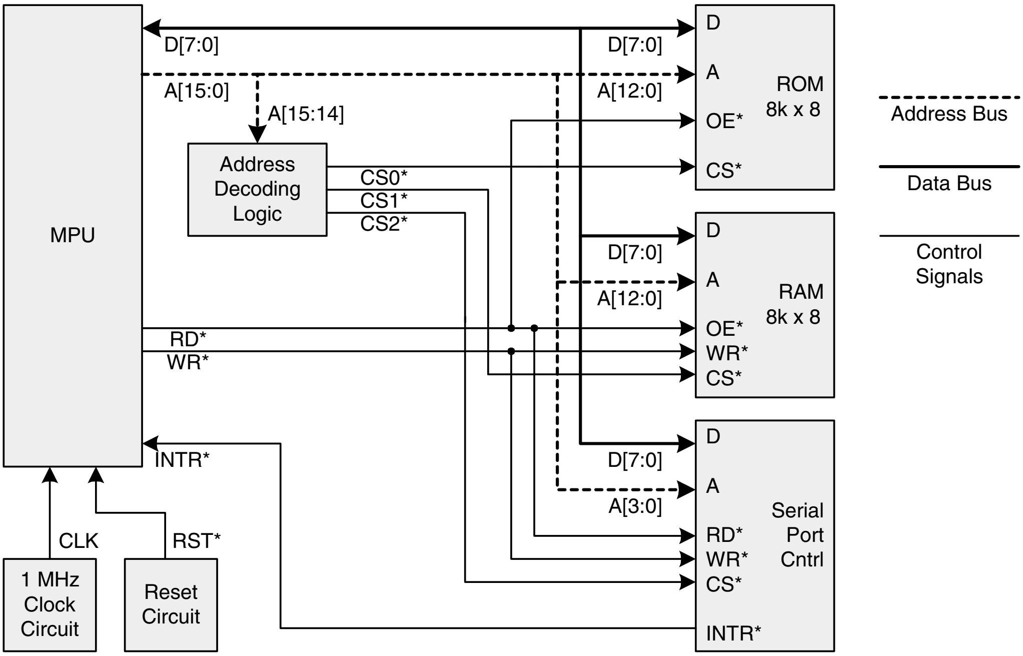


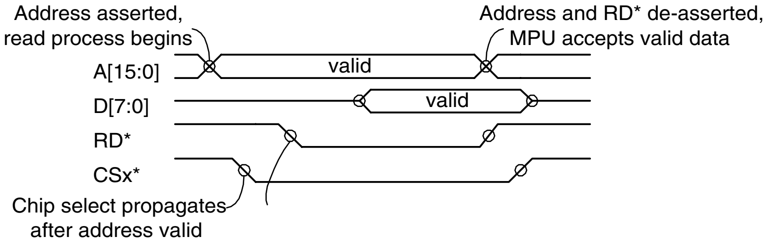
FIGURE 3.7 Eight-bit computer block diagram.

hypothetical computer are active-low, as are the control signals in most computer designs that, according to convention, have been in widespread use for the past few decades. Active-low signal names have some type of symbol as a prefix or suffix to the signal name that distinguishes them from active-high signals. Common symbols used for this purpose include #, *, –, and _. From a logical perspective, it is perfectly valid to use active-high signaling. However, because most memory and peripheral devices conform to the active-low convention, it is often easier to go along with the established convention.

While hypothetical, the microprocessor shown contains characteristics that are common in off-the-shelf eight-bit microprocessors. It contains an 8-bit data bus and a 16-bit address bus with a total address space of 64 kB. The combined MPU bus, consisting of address, data, and control signals, is asynchronous and is enabled by the assertion of read and write enable signals. When the microprocessor wants to read a location in memory, it asserts the appropriate address along with RD* and then takes the resulting value driven onto the data bus. As shown in the diagram, memory chips usually have *output enable* (OE*) signals that can be connected to a read enable. Such devices continuously decode the address bus and will emit data whenever OE* is active.

Not all 64 kB of address space is used in this computer. Address decoding logic breaks the single 64-kB space into four 16-kB regions. According to the state of A[15:14], one and only one of the chip select signals is activated. The address decoding follows the truth table shown in Table 3.1 and establishes four address ranges.

Once decoded into regions, A[13:0] provides unique address information to the memory and I/O devices connected to the MPU bus. One memory region, the upper 16 kB, is currently left unused. It may be used in the future if more memory or another I/O device is added. Each memory and I/O de-



MPU and the devices with which it is communicating. For this interface to work properly, the MPU must allow enough time for the read to occur, regardless of the specific device with which it is communicating. In other words, it must operate according to the capabilities of the slowest device—the least common denominator.

Write timing is very similar, as seen in Fig. 3.9. Again, the MPU asserts the desired address onto A[15:0], and the appropriate chip select is decoded. At the same time, the write data is driven onto D[7:0]. Once the address and data have had time to stabilize, and after allowing time for the chip select to propagate, the WR* enable signal is asserted to actually trigger the write. The WR* signal is de-asserted while data, address, and chip select are still stable so that there is no possibility of writing to a different location and corrupting data. If the WR* signal is de-asserted at the same time as the others, a race condition could develop wherein a particular device may sense the address (or data or chip select) change just prior to WR* changing, resulting in a false write to another location or to the current location with wrong data. Being an asynchronous interface, the duration of all signal assertions must be sufficient for all devices to properly execute the write.

An MPU interrupt signal is asserted by the serial port controller to enable easier programming of the serial port communication routine. Rather than having software continually poll the serial port to see if data are waiting, the controller is configured to assert INTR* whenever a new byte arrives. The MPU is then able to invoke an ISR, which can transfer the data byte from the serial port to the RAM. The interrupt also helps when transmitting data, because the speed of the typical serial port (often 9,600 to 38,400 bps) is very slow as compared to the clock speed of even a slow MPU (1 to 10 MHz). When the software wants to send a set of bytes out the serial port, it must send one byte and then wait a relatively long time until the serial port is ready for the next byte. Instead of polling in a loop between bytes, the serial port controller asserts INTR* when it is time to send the next byte. The ISR can then respond with the next byte and return control to the main program that is running at the time. Each time INTR* is asserted and the ISR responds, the ISR must be sure to clear the interrupt condition in the serial port. Depending on the exact serial port device, a read or write to a specific register will clear the interrupt. If the interrupt is not cleared before the ISR issues a return-from-interrupt, the MPU may be falsely interrupted again for the same condition.

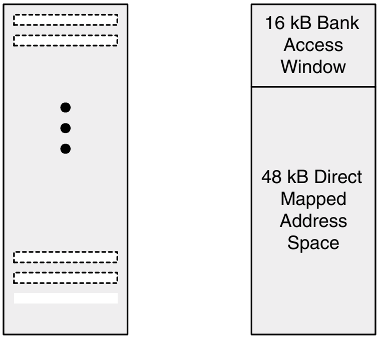
This computer contains two other functional elements: the clock and reset circuits. The 1-MHz clock must be supplied to the MPU continually for proper operation. In this example design, no other components in the computer require this clock. For fairly simple computers, this is a realistic scenario, because the buses and memory devices operate asynchronously. Many other computers, however, have synchronous buses, and the microprocessor clock must be distributed to other components in the system.

The reset circuit exists to start the MPU when the system is first turned on. Reset must be applied for a certain minimum duration after the power supply has stabilized. This is to ensure that the digital circuits properly settle to known states before they are released from reset and allowed to begin normal operation. As the computer is turned on, the reset circuit actively drives the RST* signal. Once power has stabilized, RST* is de-asserted and remains in this state indefinitely.

3.6 ADDRESS BANKING

A microprocessor's address space is normally limited by the width of its address bus, but supplemental logic can greatly expand address space, subject to certain limitations. Address banking is a technique that increases the amount of memory a microprocessor can address. If an application requires 1 MB of RAM for storing large data structures, and an 8-bit microprocessor is used with a 64-kB address space, address banking can enable the microprocessor to access the full 1 MB one small section at a time.

Address banking, also known as *paging*, takes a large quantity of memory, divides it into multiple smaller banks, and makes each bank available to the microprocessor one at a time. A *bank address register* is maintained by the microprocessor and determines which bank of memory is selected at any given time. The selected bank is accessed through a portion of the microprocessor's fixed address space, called a *window*, set aside for banked memory access. As shown in Fig. 3.10a, the upper 16 kB of address space provides direct access to one of many 16-kB pages in the larger banked memory structure. Figure 3.10b shows the logical implementation of this banked memory scheme. A



22-bit combined address is sent to the 4-MB banked memory structure: 256 pages × 16 kB per page = 4 MB. These 22 bits are formed through the concatenation of the 8-bit bank address register and 14 of the microprocessor's low-order address bits, A[13:0]. The eight bank-address bits are changed infrequently whenever the microprocessor is ready for a new page in memory. The 14 microprocessor-address bits can change each time the window is accessed.

The details of a banking scheme can be modified according to the application's requirements. The bank access window can be increased or decreased, and more or fewer pages can be defined. If an application operates on many small sets of data, a larger number of smaller pages may be suitable. If the data or software set is widely dispersed, it may be better to increase the window size as much as possible to minimize the bank address register update rate.

While address banking can greatly increase the memory available to a microprocessor, it does so with the penalties of increased access time on page switches and more complexity in managing the segmented address space. Each time the microprocessor wants to access a location in a different page, it must update the bank address register. This penalty is acceptable in some applications. However, if the application requires both consistently fast access time and large memory size, a faster, more expensive microprocessor may be re_8—Asso…f PfHox©Hfw-x[*fios3fmor x9UHfbfe x9UHfbfe x9UH mally despite being controlled by the DMAC rather than the microprocessor. Figure 3.11 shows the basic internal structure of a DMAC.

A DMA transfer can be initiated by either the microprocessor or an I/O device that contains logic to assert a request to the DMAC. DMA transfers are generally broken into two categories: peripheral/memory and memory/memory. Peripheral/memory transfers move data to a peripheral or retrieve data from a peripheral. A peripheral/memory transfer can be triggered by a DMA-aware I/O-device when it is ready to accept more outgoing data or incoming data has arrived. These are called *single-address transfers,* because the DMAC typically controls only a single address—that of the memory side of the transfer. The peripheral address is typically a fixed offset into its register set and is asserted by supporting control logic that assists in the connectivity between the peripheral and the DMAC.

DMA transfers do not have to be continuous, and they are often not in the case of a peripheral transfer. If the microprocessor sets up a DMA transfer from a serial communications controller to memory, it programs the DMAC to write a certain quantity of data into memory. However, the transfer does not begin until the serial controller asserts a DMA request indicating that data is ready. When this request occurs, the DMAC arbitrates for access to the microprocessor bus by asserting a bus request. Some time later, the microprocessor or its support logic will grant the bus to the DMAC and temporarily pause the microprocessor's bus activity. The DMAC can then transfer a single unit of data from the serial controller into memory. The unit of data transfer may be any number of bytes. When finished, the DMAC relinquishes control of the bus back to the microprocessor.

Memory/memory transfers move data from one region in memory to another. These are called *dual-address transfers,* because the DMAC controls two addresses into memory—source and destination. Memory/memory transfers are triggered by the microprocessor and can execute continuously, because the data block to be moved is ready and waiting in memory.

Even when DMA transfers execute one byte at a time, they are still more efficient than the microprocessor, because the DMAC is capable of transferring a byte or word (per the microprocessor's data bus width) in a single bus cycle rather than the microprocessor's load/store mechanism with additional overhead. There is some initial overhead in setting up the DMA transfer, so it is not efficient to use DMA for very short transfers. If the microprocessor needs to move only a few bytes, it should probably do so on its own. However, the DMAC initialization overhead is more than compensated for if dozens or hundreds of bytes are being moved.

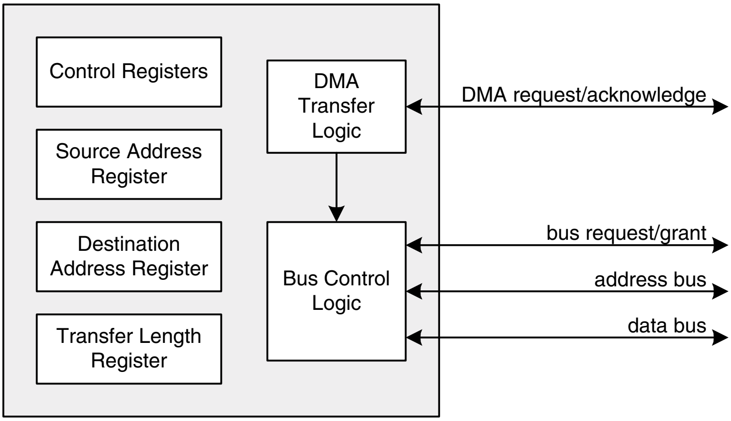


FIGURE 3.11 DMA controller block diagram.

additional memory chip. In these situations, a simple *buffered* extension of the microprocessor bus may suffice. A buffer, in this context, is an IC that passes data from one set of pins to another, thereby electrically separating two sections of a bus. As shown in Fig. 3.12, a buffer can extend a microprocessor bus so that its logical functionality remains unchanged, but its electrical characteristics are enhanced to provide connectivity across a greater distance (to a multichip memory expansion module). A unidirectional address buffer extends the address bus from the microprocessor to expansion memory devices. A bidirectional data buffer extends the bus away from the microprocessor on writes and toward the microprocessor on reads. The direction of the data buffer is controlled according to the state of read/write enable signals generated by the microprocessor.

More complex memory structures may contain dedicated memory control logic that sits between the microprocessor and the actual memory devices. Expanding such a memory architecture is generally accomplished by augmenting the "back-side" memory device bus as shown in Fig. 3.13 rather than by adding additional controllers onto an extended microprocessor bus. Such an expansion scheme may or may not require buffers, depending on the electrical characteristics of the bus in question.

I/O buses may also be direct extensions of the microprocessor bus. The original expansion bus in the IBM PC, developed in the early 1980s, is essentially an extended Intel 8088 microprocessor bus that came to be known as the *Industry Standard Architecture* (ISA) bus. Each I/O card on the ISA bus is mapped in a unique address range in the microprocessor's memory. Therefore, when software wants to read or write a register on an I/O card, it simply performs an access to the desired location. The ISA bus added a few features beyond the raw 8088 bus, including DMA and variable *wait states* for slow I/O devices. A wait state results when a device cannot immediately respond to the microprocessor's request and asserts a signal to stretch the access so that it can respond properly.

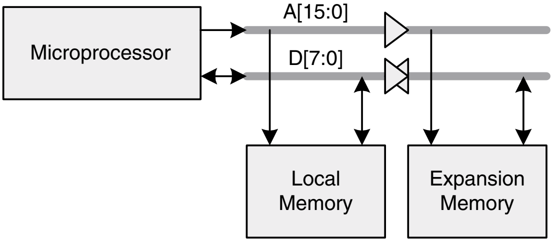


FIGURE oHMx+VfiPC-V92HPC+P9xT=0CC+9PC2HDoHMM+VfiPV9--2HHH+-PPV92btGo'T=Tfi2HHP-9PV92Cs1R(9V20f

Direct extensions such as the ISA bus are fairly easy to implement and serve well in applications where I/O response time does not unduly restrict microprocessor throughput. As computers have gotten faster, the throughput of microprocessors has rapidly outstripped the response times of all but the fastest I/O devices. In comparison to a modern microprocessor, a hard-disk controller is rather slow, with response times measured in microseconds rather than nanoseconds. Additionally, as bus signals become faster, the permissible length of interconnecting wires decreases, limiting their expandability. These and other characteristics motivate the decoupling of the microprocessor's local bus from the computer's I/O bus.

An I/O bus can be decoupled from the microprocessor bus by inserting an intermediate bus controller between them that serves as an interface, or translator, between the two buses. Once the buses are separated, activity on one bus does not necessarily obstruct activity on the other. If the microprocessor wants to write a block of data to a slow device, it can rapidly transfer that data to the bus controller and then continue with other operations at full speed while the controller slowly transfers the data to the I/O device. This mechanism is called a *posted-write,* because the bus controller allows the microprocessor to complete, or *post*, its write before the write actually completes. Separate buses also open up the possibility of multiple microprocessors or logic elements performing I/O operations without conflicting with the central microprocessor. In a multimaster system, a specialized DMA controller can transfer data between two peripherals such as disk controllers while the microprocessor goes about its normal business.

The *Peripheral Component Interconnect* (PCI) bus is the industry-standard follow-on to the ISA bus, and it implements such advanced features as posted-writes, multiple-masters, and multiple bus segments. Each PCI bus segment is separated from the others via a PCI bridge chip. Only traffic that must travel between buses crosses a bridge, thereby reducing congestion on individual PCI bus segments. One segment can be involved in a data transfer between two devices without affecting a simultaneous transfer between two other devices on a different segment. These performance-enhancing features do not come for free, however. Their cost is manifested by the need for dedicated PCI control logic in bridge chips and in the I/O devices themselves. It is generally simpler to implement an I/O device that is directly mapped into the microprocessor's memory space, but the overall performance of the computer may suffer under demanding applications.

3.9 ASSEMBLY LANGUAGE AND ADDRESSING MODES

With the hardware ready, a computer requires software to make it more than an inactive collection of components. Microprocessors fetch instructions from program memory, each consisting of an opcode and, optionally, additional operands following the opcode. These opcodes are binary data that are easy for the microprocessor to decode, but they are not very readable by a person. To enable a programmer to more easily write software, an instruction representation called *assembly language* was developed. Assembly language is a low-level language that directly represents each binary opcode with a human-readable text mnemonic. For example, the mnemonic for an unconditional branch-to-subroutine instruction could be `BSR`. In contrast, a high-level language such as C++ or Java contains more complex logical expressions that may be automatically converted by a compiler to dozens of microprocessor instructions. Assembly language programs are assembled, rather than compiled, into opcodes by directly translating each mnemonic into its binary equivalent.

Assembly language also makes programming easier by enabling the usage of text labels in place of hard-coded addresses. A subroutine can be named `FOO`, and when `BSR FOO` is encountered by the assembler, a suitable branch target address will be automatically calculated in place of the label `FOO`. Each type of assembler requires a slightly different format and syntax, but there are general assembly language conventions that enable a programmer to quickly adapt to specific implementations

- *Immediate addressing*

trary entry in a data table, the software can load the address of that entry into a microprocessor register and then perform an indirect access using that register as a pointer. Some microprocessors place constraints on which registers can be used as references for indirect addressing. In the case of a 6800 microprocessor, `LDAA (ACCB)` is not actually a supported operation but serves as a syntactical example for purposes of discussion.

- *Indexed addressing* is a close relative (no pun intended) of indirect addressing, because it also refers to an address contained in another register. However, indexed addressing also specifies an offset, or index, to be added to that register base value to generate the final operand address: base + offset = final address. Some microprocessors allow general accumulator registers to be used as base-address registers, but others, such as the 6800, provide special *index registers* for this purpose. In many 8-bit microprocessors, a full 16-bit address cannot be obtained from an 8-bit accumulator serving as the base address. Therefore, one or more separate index registers are present for the purpose of indexed addressing. In contrast, many 32-bit microprocessors are able to specify a full 32-bit address with any general-purpose register and place no limitations on which register serves as the index register. Indexed addressing builds upon the capabilities of indirect addressing by enabling multiple address offsets to be referenced from the same base address. `LDAA (X+$20)` would tell the microprocessor to add $20 to the index register, X, and use the resulting address to fetch data to be loaded into ACCA. One simple example of using indexed addressing is a subroutine to add a set of four numbers located at an arbitrary location in memory. Before calling the subroutine, the main program can set an index register to point to the table of numbers. Within the subroutine, four individual addition instructions use the indexed addressing mode to add the locations X+0, X+1, X+2, and X+3. When so written, the subroutine is flexible enough to be used for any such set of numbers. Because of the similarity of indexed and indirect addressing, some microprocessors merge them into a single mode and obtain indirect addressing by performing indexed addressing with an index value of zero.

The six conceptual addressing modes discussed above represent the various logical mechanisms that a microprocessor can employ to access data. It is important to realize that each individual microprocessor applies these addressing modes differently. Some combine multiple modes into a single mode (e.g., indexed and indirect), and some will create multiple submodes out of a single mode. The exact variation depends on the specifics of an individual microprocessor's architecture.

With the various addressing modes modifying the specific opcode and operands that are presented to the microprocessor, the benefits of using assembly language over direct binary values can be observed. The programmer does not have to worry about calculating branch target addresses or resolving different addressing modes. Each mnemonic can map to several unique opcodes, depending on the addressing mode used. For example, the `LDAA` instruction in Fig. 3.14 could easily have used *extended* addressing by specifying a full 16-bit address at which the ASCII transmit-value is located. Extended addressing is the 6800's mechanism for specifying a 16-bit direct address. (The 6800's direct addressing involves only an eight-bit address.) In either case, the assembler would determine the correct opcode to represent `LDAA` and insert the correct binary values into the memory dump. Additionally, because labels are resolved each time the program is assembled, small changes to the program can be made that add or remove instructions and labels, and the assembler will automatically adjust the resulting addresses accordingly.

Programming in assembly language is different from using a high-level language, because one must think in smaller steps and have direct knowledge about the microprocessor's operation and architecture. Assembly language is processor-specific instead of generic, as with a high-level language. Therefore, assembly language programming is usually restricted to special cases such as boot code or routines in which absolute efficiency and performance are demanded. A human programmer will usually be able to write more efficient assembly language than a high-level language compiler

CHAPTER 4
Memory

Memory is as fundamental to computer architecture as any other element. The ability of a system's memory to transact the right quantity of data in the right span of time has a substantial impact on how that system fulfills its design goals. Digital engineers struggle with innovative ways to improve memory density and bandwidth in a way that is tailored to a specific application's performance and cost constraints.

Knowledge of prevailing memory technologies' strengths and weaknesses is a key requirement for designing digital systems. When memory architecture is chosen that complements the rest of the system, a successful design moves much closer to fruition. Conversely, inappropriate memory architecture can doom a good idea to the engineering doldrums of impracticality brought on by artificial complexity.

This chapter provides an introduction to various solid-state memory technologies and explains how they work from an internal structural perspective as well as an interface timing perspective. A memory's internal structure is important to an engineer, because it explains why that memory might be more suited for one application over another. Interface timing is where the rubber meets the road, because it defines how other elements in the system can access memory components' contents. The wrong interface on a memory chip can make it difficult for external logic such as a microprocessor to access that memory and still have time left over to perform the necessary processing on that data.

Basic memory organization and terminology are introduced first. This is followed by a discussion of the prevailing read-only memory technologies: EPROM, flash, and EEPROM. Asynchronous SRAM and DRAM technologies, the foundations for practically all random-access memories, are presented next. These asynchronous RAMs are no longer on the forefront of memory technology but still find use in many systems. Understanding their operation not only enables their application, it also contributes to an understanding of the most recent synchronous RAM technologies. (High-per-
[redacted] rmance synchronous memories are discussed later in the book.) The chapter concludes with a dis-
ion of two types of specialty memories: multiport RAMs and FIFOs. Multiport RAMs and
s are found in many applications where memory serves less as a storage element and more as a
nications channel between distinct logic blocks.

SIFICATIONS

sors require memory resources in which to store programs and data. Memory can be
o two broad categories: volatile and nonvolatile. Volatile memory loses its contents
s turned off. Nonvolatile memory retains its contents indefinitely, even when there is no
t. Nonvolatile memory can be used to hold the boot code for a computer so that the mi-
can have a place to get started. Once the computer begins initializing itself from non-
ory, volatile memory is used to store dynamic variables, including the stack and other
t may be loaded from a disk drive. Figure 4.1 shows that a general memory device con-
storage array, address-decode logic, input/output logic, and control logic.

Once programmed, the charge on the floating gate cannot be removed electrically. UV photons cause the dielectric to become slightly conductive, allowing the floating gate's charge to gradually drain away to its unprogrammed state. This UV erasure feature is the reason why many EPROMs are manufactured in ceramic packages with transparent quartz windows directly above the silicon die. These ceramic packages are generally either DIPs or PLCCs and are relatively expensive. In the late 1980s it became common for EPROMs to be manufactured in cheaper plastic packages without transparent windows. These EPROM devices are rendered one-time programmable, or OTP, because it is impossible to expose the die to UV light. OTP devices are attractive, because they are the least expensive nonmask ROM technology and provide a manufacturer with the flexibility to change software on the assembly line by using a new data image to program EPROMs.

The industry standard EPROM family is the 27xxx, where the "xxx" indicates the chip's memory capacity in kilobits. The 27256 and 27512 are very common and easily located devices. Older parts include the 2708, 2716, 2732, 2764, and 27128. There are also newer, higher-density EPROMs such as the 27010, 27020, and 27040 with 1 Mb, 2 Mb, and 4 Mb densities, respectively. 27xxx EPROM devices are most commonly eight bits wide (a 27256 is a 32,768 × 8 EPROM). Wider data words, such as 16 or 32 bits, are available but less common.

Older members of the 27xxx family, such as early NMOS 2716 and 2732 devices, required 21-V programming voltages, consumed more power, and featured access times of between 200 and 450 ns. Newer CMOS devices are designated 27Cxxx, require a 12-V programming voltage, consume less power, and have access times as fast as 45 ns, depending on the manufacturer and device density.

EPROMs are very easy to use because of their classic asynchronous interface. In most applications, the EPROM is treated like a ROM, so writes to the device are not an issue. Two programming control pins, V_{PP} and PGM*, serve as the high-voltage source and program enable, respectively. These two pins can be set to inactive levels and forgotten. What remains are a chip enable, CE*, an output enable, OE*, an address bus, and a data output bus as shown in Fig. 4.3, using a 27C64 (8K × 8) as an example.

When CE* is inactive, or high, the device is in a powered-down mode in which it consumes the least current—measured in microamps due to the quiescent nature of CMOS logic. When CE* and OE* are active simultaneously, D[7:0] follows A[12:0] subject to the device's access time, or propagation delay. This read timing is shown in Fig. 4.4.

When OE* is inactive, the data bus is held in a high-impedance state. A certain time after OE* goes active, t_{OE}, the data word corresponding to the given address is driven—assuming that A1 has been stable for at least t_{ACC}. If not, t_{ACC} will determine how soon D1 is available rather than t_{OE}. While OE* is active, the data bus transitions t_{ACC} ns after the address bus. As soon as OE* is removed, the data bus returns to a high-impedance state after t_{OEZ}.

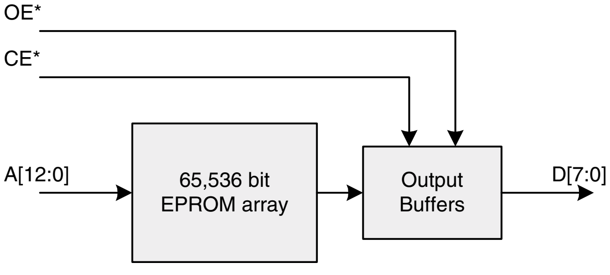


FIGURE 4.3 27C64 block diagram.

Many microprocessors are able to directly interface to an EPROM via this asynchronous bus because of its ubiquity. Most eight-bit microprocessors have buses that function solely in this asynchronous mode. In contrast, some high-performance 32-bit microprocessors may initially boot in a low-speed asynchronous mode and then configure themselves for higher performance operation after retrieving the necessary boot code and initialization data from the EPROM.

4.3 FLASH MEMORY

Flash memory captured the lion's share of the nonvolatile memory market from EPROMs in the 1990s and holds a dominant position as the industry leader to this day. Flash is an enhanced EPROM that can both program and erase electrically without time-consuming exposure to UV light, and it has no need for the associated expensive ceramic and quartz packaging. Flash does cost a small amount more to manufacture than EPROM, but its more flexible use in terms of electronic erasure more than makes up for a small cost differential in the majority of applications. Flash is found in everything from cellular phones to automobiles to desktop computers to solid-state disk drives. It has enabled a whole class of flexible computing platforms that are able to upgrade their software easily and "on the fly" during normal operation. Similar to EPROMs, early flash devices required separate programming voltages. Semiconductor vendors quickly developed single-supply flash devices that made their use easier.

A flash bit structure is very similar to that of an EPROM. Two key differences are an extremely thin dielectric between the floating gate and the silicon substrate and the ability to apply varying bias voltages to the source and control gate. A flash bit is programmed in the same way that an EPROM bit is programmed—by applying a high voltage to the control gate. Flash devices contain internal voltage generators to supply the higher programming voltage so that multiple external voltages are not required. The real difference appears when the bit is erased electrically. A rather complex quantum-mechanical behavior called *Fowler-Nordheim tunneling* is exploited by applying a negative voltage to the control gate and a positive voltage to the MOSFET's source as shown in Fig. 4.5.

The combination of the applied bias voltages and the thin dielectric causes the charge on the floating gate to drain away through the MOSFET's source. Flash devices cannot go through this program/erase cycle indefinitely. Early M—Undefinpo*SoGAT

Bg*Arf÷f0D—8©w m——Hfm…—A E8©8—8Hc8fi8—AxfGw88—xwUfaalAechOSFET'xCOA ©s x9

Modern flash devices require only a single supply voltage and contain on-chip circuitry to create the nonstandard programming and erasure voltages required by the memory array. Control logic determines which block is placed into erase or program mode at any given time as requested by the microprocessor with a predefined flash control algorithm. AMD's algorithm consists of six special write transactions to the flash: two unlock cycles, a setup command, two more unlock cycles, and the specific erase command. This sequence is detailed in Table 4.1. If interrupted, the sequence must be restarted to ensure integrity of the command.

For a whole-chip erase, the address/data in cycle 6 is 0x555/0x10. For a single-sector erase, the address/data in cycle 6 is the sector address/0x30. Multiple erase commands o8Aomma—E88wdDnfbex[*fio*S

SRAM implementations require six transistors per bit of memory: two transistors for each inverter and the two pass transistors. Some implementations use only a single transistor per inverter, requiring only four transistors per bit.

Discrete asynchronous SRAM devices have been around for decades. In the 1980s, the 6264 and 62256 were manufactured by multiple vendors and used in applications that required simple RAM architectures with relatively quick access times and low power consumption. The 62xxx family is numbered according to its density in kilobits. Hence, the 6264 provides 65,536 bits of RAM arranged as 8k × 8. The 62256 provides 262,144 bits of RAM arranged as 32k × 8. Being manufactured in CMOS technology and not using a clock, these devices consume very little power and draw only microamps when not being accessed.

The 62xxx family pin assignment is virtually identical to that of the 27xxx EPROM family, enabling system designs where either EPROM or SRAM can be substituted into the same location with only a couple of jumpers to set for unique signals such as the program-enable on an EPROM or write-enable on an SRAM. Like an EPROM or basic flash device, asynchronous SRAMs have a simple interface consisting of address, data, chip select, output enable, and write enable. This interface is shown in Fig. 4.8.

Writes are performed whenever the WE* signal is held low. Therefore, one must ensure that the desired address and data are stable before asserting WE* and that WE* is removed while address and data remain stable. Otherwise, the write may corrupt an undesired memory location. Unlike an EPROM, but like flash, the data bus is bidirectional during normal operation. The first two transactions shown are writes as evidenced by the separate assertions of WE* for the duration of address and data stability. As soon as the writes are completed, the microprocessor should release the data bus to the high-impedance state. When OE* is asserted, the SRAM begins driving the data bus and the output reflects the data contents at the locations specified on the address bus.

Asynchronous SRAMs are available with access times of less than 100 ns for inexpensive parts and down to 10 ns for more expensive devices. Access time measures both the maximum delay between a stable read address and its corresponding data and the minimum duration of a write cycle. Their ease of use makes them suitable for small systems where megabytes of memory are not required and where reduced complexity and power consumption are key requirements. Volatile memory doesn't get any simpler than asynchronous SRAM.

Prior to the widespread availability of flash, many computer designs in the 1980s utilized asynchronous SRAM with a battery backup as a means of implementing nonvolatile memory for storing configuration information. Because an idle SRAM draws only microamps of current, a small battery can maintain an SRAM's contents for several years while the main power is turned off. Using SRAM in this manner has two distinct advantages over other technologies: writes are quick and easy, because there are no complex EEPROM or flash programming algorithms, and there is no limit to the number of write cycles performed over the life of the product. The downsides to this approach are a lack of security for protecting valuable configuration information and the need for a battery to

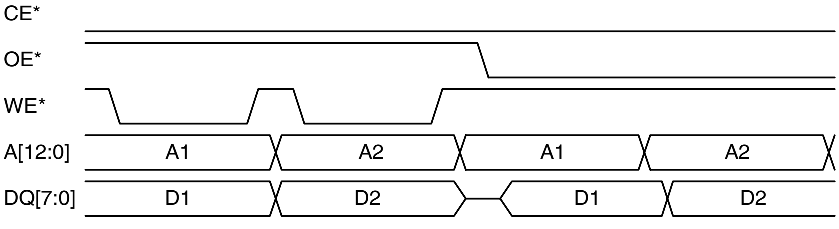


FIGURE 4.8 62xxx SRAM interface.

maintain the memory contents. Requiring a battery increases the complexity of the system and also begs the question of what happens when the battery wears out. In the 1980s, it was common for a PC's BIOS configuration to be stored in battery-backed CMOS SRAM. This is how terms like "the CMOS" and "CMOS setup" entered the lexicon of PC administration.

SRAM is implemented not only as discrete memory chips but is commonly found integrated within other types of chips, including microprocessors. Smaller microprocessors or *microcontrollers* (microprocessors integrated with memory and peripherals on a single chip) often contain a quantity of on-board SRAM. More complex microprocessors may contain on-chip data caches implemented with SRAM.

4.6 ASYNCHRONOUS DRAM

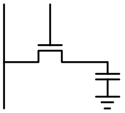


FIGURE 4.9 DRAM bit structure.

SRAM may be the easiest volatile memory to use, but it is not the least expensive in significant densities. Each bit of memory requires between four and six transistors. When millions or billions of bits are required, the complexity of all those transistors becomes substantial. Dynamic RAM, or DRAM, takes advantage of a very simple yet fragile storage component: the capacitor. A capacitor holds an electrical charge for a limited amount of time as the charge gradually drains away. As seen from EPROM and flash devices, capacitors can be made to hold charge almost indefinitely, but the penalty for doing so is significant complexity in modifying the storage element. Volatile memory must be both quick to access and not be subject to write-cycle limitations—both of which are restrictions of nonvolatile memory technologies. When a capacitor is designed to have its charge quickly and easily manipulated, the downside of rapid discharge emerges. A very efficient volatile storage element can be created with a capacitor and a single transistor as shown in Fig. 4.9, but that capacitor loses its contents soon after being charged. This is where the term *dynamic* comes from in DRAM—the memory cell is indeed dynamic under steady-state conditions. The solution to this problem of solid-state amnesia is to periodically refresh, or update, each DRAM bit before it completely loses its charge.

As with SRAM, the pass transistor enables both reading and writing the state of the storage element. However, a single capacitor takes the place of a multitransistor latch. This significant reduction in bit complexity enables much higher densities and lower per-bit costs when memory is implemented in DRAM rather than SRAM. This is why main memory in most computers is implemented using DRAM. The trade-off for cheaper DRAM is a degree of increased complexity in the memory control logic. The number one requirement when using DRAM is periodic refresh to maintain the contents of the memory.

DRAM is implemented as an array of bits with rows and columns as shown in Fig. 4.10. Unlike SRAM, EPROM, and flash, DRAM functionality from an external perspective is closely tied to its row and column organization.

SRAM is accessed by presenting the complete address simultaneously. A DRAM address is presented in two parts: a row and a column address. The row and column addresses are multiplexed onto the same set of address pins to reduce package size and cost. First the row address is loaded, or strobed, into the row address latch via *row address strobe*, or RAS*, followed by the column address with *column address strobe*, or CAS*. Read data propagates to the output after a specified access time. Write data is presented at the same time as the column address, because it is the column strobe that actually triggers the transaction, whether read or write. It is during the column address phase that WE* and OE* take effect.

of the transaction. Some time later, the read data is made available on the data bus. After waiting for a sufficient time for the DRAM to return the read data, the memory controller removes RAS* and CAS* to terminate the transaction.

Basic writes are similar to single reads as shown in Fig. 4.12. Again, CE* is assumed to be held active, and, being a write, OE* is assumed to be held inactive throughout the transaction.

Like a read, the write transaction begins by loading the row address. From this it is apparent that there is no particular link between loading a row address and performing a read or a write. The identity of the transaction is linked to the falling edge of CAS*, when WE* is asserted at about the same time that the column address and write data are asserted. DRAM chips require a certain setup and hold time for these signals around the falling edge of CAS*. Once the timing requirements are met, address can be deasserted prior to the rising edge of CAS*.

A read/write hybrid transaction, called a *read-modify-write*, is also supported to improc bD

Software does occasionally branch back and forth in its memory space. Yet, on the whole, software moves through portions of memory in a linear fashion. FPM devices enable a DRAM controller to load a row-address in the normal manner using RAS* and then perform multiple CAS* transactions using the same row-address. Therefore, DRAMs end their transaction cycles with the rising edge of RAS*, because they cannot be sure if more reads or writes are coming until RAS* rises, indicating that the current row-address can be released.

FPM technology, in turn, gave way to *extended-data out* (EDO) devices that extend the time read data is held valid. Unlike its predecessors, an EDO DRAM does not disable the read data when CAS* rises. Instead, it waits until either the transaction is complete (RAS* rises), OE* is deasserted,

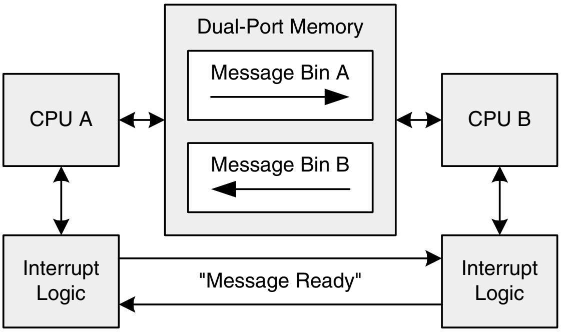


FIGURE 4.17 Dual microprocessor message passing architecture.

4.8 THE FIFO

The memory devices discussed thus far are essentially linear arrays of bits surrounded by a minimal quantity of interface logic to move bits between the port(s) and the array. *First-in-first-out* (FIFO) memories are special-purpose devices that implement a basic queue structure that has broad application in computer and communications architecture. Unlike other memory devices, a typical FIFO has two unidirectional ports without address inputs: one for writing and another for reading. As the name implies, the first data written is the first read, and the last data written is the last read. A FIFO is not a random access memory but a sequential access memory. Therefore, unlike a conventional memory, once a data element has been read once, it cannot be read again, because the next read will return the next data element written to the FIFO. By their nature, FIFOs are subject to *overflow* and *underflow* conditions. Their finite size, often referred to as *depth*, means that they can fill up if reads do not occur to empty data that has already been written. An overflow occurs when an attempt is made to write new data to a full FIFO. Similarly, an empty FIFO has no data to provide on a read request, which results in an underflow.

A FIFO is created by surrounding a dual-port memory array—generally SRAM, but DRAM could be made to work as well for certain applications—with a write pointer, a read pointer, and control logic as shown in Fig. 4.18.

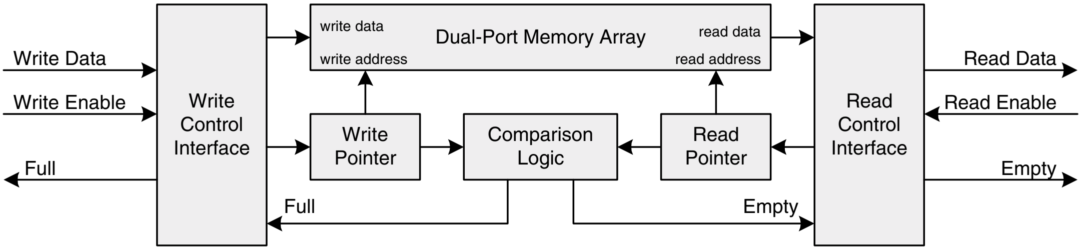


FIGURE 4.18 Basic FIFO architecture.

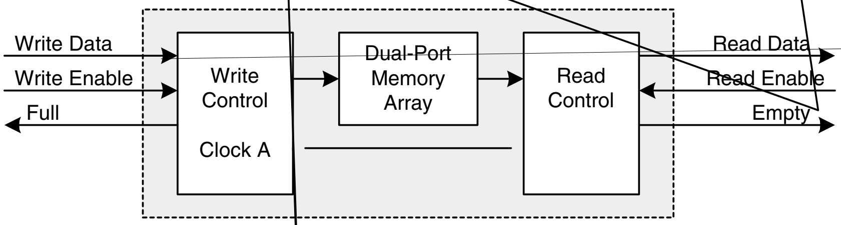
A FIFO is not addressed in a linear fashion; rather, it is made to form a continuous ring of memory that is addressed by the two internal pointers. The fullness of the FIFO is determined not by the absolute values of the pointers but by their relative values. An empty FIFO begins with its read and write pointers set to the same value. As entries are written, the write pointer increments. As entries are read, the read pointer increments as well. If the read pointer ever catches up to the write pointer such that the two match, the FIFO is empty again. If the read pointer fails to advance, the write pointer will eventually wrap around the end of the memory array and become equal to the read pointer. At this point, the FIFO is full and cannot accept any more data until reading resumes. Full and empty flags are generated by the FIFO to provide status to the writing and reading logic. Some FIFOs contain more detailed fullness status, such as signals that represent programmable fullness thresholds.

The interfaces of a FIFO can be asynchronous (no clock) or synchronous (with a clock). If synchronous, the two ports can be designed to operate with a common clock or different clocks. Although older asynchronous FIFOs are still manufactured, synchronous FIFOs are now more common. Synchronous FIFOs have the advantage of improved interface timing, because flops placed at a device's inputs and outputs reduce timing requirements to the familiar setup, hold, and clock-to-out specifications. Without such a registered interface, timing specifications become a function of the device's internal logic paths.

One common role that a FIFO fills is in clock domain crossing. In such an application, there is a need to communicate a series of data values from a block of logic operating on one clock to another block operating on a different clock. Exchanging data between clock domains requires special attention, because there is normally no way to perform a conventional timing analysis across two different clocks to guarantee adequate setup and hold times at the destination flops. Either an asynchronous FIFO or a dual-clock synchronous FIFO can be used to solve this problem, as shown in Fig. 4.19.

The dual-port memory at the heart of the FIFO is an asynchronous element that can be accessed by the logic operating in either clock domain. A dual-clock synchronous FIFO is designed to handle arbitrary differences in the clocks between the two halves of the device. When one or more bytes are written on clock A, the write-pointer information is carried safely across to the clock B domain within the FIFO via inter-clock domain synchronization logic. This enables the read-control interface to determine that there is data waiting to be read. Logic on clock B can read this data long after it has been safely written into the memory array and allowed to settle to a stable state.

Another common application for a FIFO is rate matching where a particular data source is bursty and the data consumer accepts data at a more regular rate. One example is a situation where a sequence of data is stored in DRAM and needs to be read out and sent over a communications interface one byte at a time. The DRAM is shared with a CPU that competes with the communications interface for memory bandwidth. It is known that DRAMs are most efficient when operated in a page-mode burst. Therefore, rather than perform a complete read-transaction each time a single byte



is needed for the communications interface, a burst of data can be read and stored in a FIFO. Each time the interface is ready for a new byte, it reads it from the FIFO. In this case, only a single-clock FIFO is required, because these devices operate on a common clock domain. To keep this process running smoothly, control logic is needed to watch the state of the FIFO and perform a new burst read from DRAM when the FIFO begins to run low on data. This scheme is illustrated in Fig. 4.20.

For data-rate matching to work properly, the average bandwidth over time of the input and output ports of the FIFO must be equal, because FIFO capacity is finite. If data is continuously written faster than it can be read, the FIFO will eventually overflow and lose data. Conversely, if data is continuously read faster than it can be written, the FIFO will underflow and cause invalid bytes to be inserted into the outgoing data stream. The depth of a FIFO indicates how large a read/write rate disparity can be tolerated without data loss. This disparity is expressed as the product of rate mismatch and time. A small mismatch can be tolerated for a longer time, and a greater rate disparity can be tolerated for a shorter time.

In the rate-matching example, a large rate disparity of brief duration is balanced by a small rate disparity of longer duration. When the DRAM is read, a burst of data is suddenly written into the FIFO, creating a temporarily large rate disparity. Over time, the communications interface reads one byte at a time while no writes are taking place, thereby compensating with a small disparity over time.

DRAM reads to refill the FIFO must be carefully timed to simultaneously prevent overflow and underflow conditions. A threshold of FIFO fullness needs to be established below which a DRAM read is triggered. This threshold must guarantee that there is sufficient space available in the FIFO to accept a full DRAM burst, avoiding an overflow. It must also guarantee that under the worst-case response time of the DRAM, enough data exists in the FIFO to satisfy the communications interface, avoiding an underflow. In most systems, the time between issuing a DRAM read request and actually getting the data is variable. This variability is due to contention with other requesters (e.g., the CPU) and waiting for overhead operations (e.g., refresh) to complete.

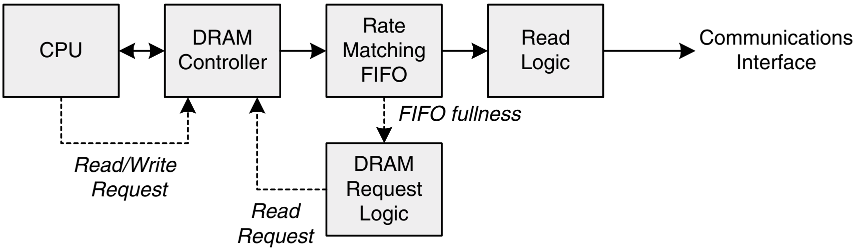


FIGURE 4.20 Synchronous FIFO application: data rate matching.

CHAPTER 5
Serial Communications

Serial communication interfaces are commonly used to exchange data with other computers. Serial interfaces are ubiquitous, because they are economical to implement over long distances as a result of their requirement of relatively few wires. Many types of serial interfaces have been developed, with speeds ranging to billions of bits per second. Regardless of the bit rate, serial communication interfaces share many common traits. This chapter introduces the fundamentals of serial communication in the context of popular data links such as RS-232 and RS-485 in which bandwidths and components lend themselves to basic circuit fabrication techniques.

The chapter first deals with the basic parallel-to-serial-to-parallel conversion process that is at the heart of all serial communication. Wide buses must be serialized at the transmitter and reconstructed at the receiver. Techniques for accomplishing this vary with the specific type of data link, but basic concepts of framing and error detection are universal.

Two widely deployed point-to-point serial communication standards, RS-232 and RS-422, are presented, along with the standard ASCII character set, to see how theory meets practice. Standards are important to communications in general because of the need to connect disparate equipment. ASCII is one of the most fundamental data representation formats with global recognition. RS-232 has traditionally been found in many digital systems, because it is a reliable standard. Understanding RS-232, its relative RS-422, and ASCII enables an engineer to design a communication interface that can work with an almost infinite range of complementary equipment ranging from computers to modems to off-the-shelf peripherals.

Systems may require more advanced communication schemes to enable data exchange between many nodes. Networks enable such communication and can range in complexity according to an application's requirements. Networking adds a new set of fundamental concepts on top of basic serial communication. Topics including network topologies and packet formats are presented to explain how networks function at a basic hardware and software level. Once networking fundamentals have been discussed, the RS-485 standard is introduced to show how a simple and fully functional network can be constructed. A complete network design example using RS-485 is offered with explanations of why various design points are included and how they contribute to the network's overall operation.

The chapter closes with a presentation of small-scale networking employed within a digital system to economically connect peripherals to a microprocessor. Interchip networks are of such narrow scope that they are usually not referred to as networks, but they can possess many fundamental properties of a larger network. Peripherals with low microprocessor bandwidth requirements can be connected using a simple serial interface consisting of just a few wires, as compared to the full complexity of a parallel bus.

Data	Even Parity	Odd Parity
0xA0 = 10100000	0	1

FIGURE 5.2 Odd and even parity.

shaking involves a receiver driving a ready signal to the transmitter. The transmitter sends data only when the receiver signals that it is ready. UARTs may support hardware handshaking. Any software handshaking is the responsibility of the UART control program.

Software handshaking works by transmitting special binary codes that either pause or resume the opposite end as it sends data. *XON/XOFF* handshaking is a common means of implementing software flow control. When one end of the link is ready to accept data, it transmits a standard character called XON (0x11) to the opposite device. When the receiver has filled a buffer and is unable to accept more data, an XOFF character (0x13) is transmitted. It is by good behavior that most flow control schemes work: the device that receives an XOFF must respect the signal and pause its transmission until an XON is received. It is not uncommon to see an XON/XOFF setting in certain serial terminal configurations.

A generic UART is shown in Fig. 5.3. The UART is divided into three basic sections: CPU interface, transmitter, and receiver. The CPU interface contains various registers to configure parity, bit rate, handshaking, and interrupts. UARTs usually provide three parity options: none, even, and odd. Bit rate is selectable well by programming an internal counter to arbitrarily divide an external reference clock. The range of usable bit clocks may be from several hundred bits per second to over 100 kbps.

Interrupts are used to inform the CPU when a new byte has been received and when a new byte is ready to be transmitted. This saves the CPU from having to constantly poll the UART's status registers for this information. However, UARTs provide status bits to aid in interrupt status reporting, so a simple serial driver program could operate by polling rather than implementing an interrupt service routine. Aside from general control and status registers, the CPU interface provides access to transmit and receive buffers so that data can be queued for transmission and retrieved upon arrival. Depending on the UART, these buffers may be only one byte each, or they may be several bytes implemented as a small FIFO. Typically, these serial ports run slow enough to not require deep buffers, because even a slow CPU can easily respond to a transmit/receive event before the data link underruns the transmit buffer or overruns the receive buffer.

The transmit section implements a parallel-to-serial shift register, parity generator, and framing logic. UARTs support framing with a start bit and one or two stop bits where the start bit is a logic 0 and stop bits are logic 1s. It is also common to transmit data LSB first. With various permutations of framing options, parity protection, and seven or eight data bits, standard configuration notation is of the form <parity:N/E/O>-<width:8/7>-<stop-bits:1/2>. For example, N-8-1 represents no parity, 8 data bits, and 1 stop bit. E-8-2 represents even parity, 8 data bits, and 2 stop bits. To help understand the format of bytes transmitted by a UART, consider Fig. 5.4. Here, two data bytes are transmitted: 0xA0 and 0x67. Keep in mind that the LSB is transmitted first.

Receiving the serial data is a bit trickier than transmitting it, because there is no clock accompanying the data with which the data can be sampled. This is where the asynchronous terminology in the UART acronym comes from. The receiver contains a clock synchronization circuit that detects the start-bit and establishes a timing reference point from which all subsequent bits in the byte will be sampled. This reference point is created using a higher-frequency receive clock. Rather than running the receiver at 1x the bit rate, it may be run at 16x the bit rate. Now the receive logic can decompose a bit into 16 time units and slide a 16-clock window according to where the start bit is observed. It is advantageous to sample each subsequent bit halfway through its validity window for maximum timing margin on either side of the sampling event. This allows maximum flexibility for settling time around the edges of the electrical pulse that defines each bit.

Consider the waveform in Fig. 5.5. When the start bit is detected, the sampling window is reset, and a sampling point halfway through is established. Subsequent bits can have degraded rising and falling edges without causing the receiver to sample an incorrect logic level.

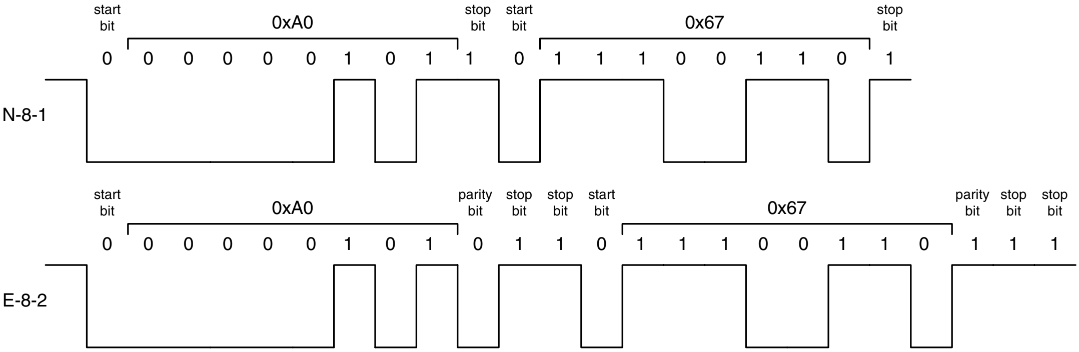
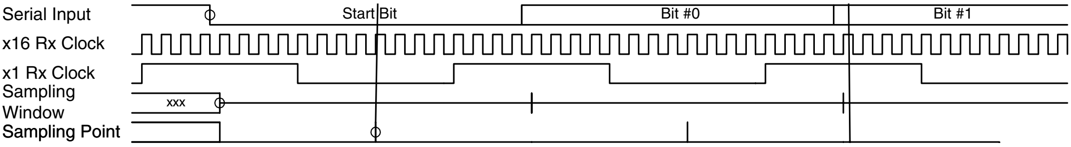


FIGURE 5.4 Common byte framing formats.



5.3 ASCII DATA REPRESENTATION

Successful communication requires standardized data representation so that people and computers around the world can share the same information. Alphanumeric characters are represented by a seven-bit standard representation known as the *American Standard Code for Information Interchange*, or ASCII. ASCII also includes punctuation marks and invisible control codes used to help in the display and transfer of data. ASCII was first published in 1968 by the *American National Standards Institute*, or ANSI. The original ASCII standard lacked provisions for many commonly used grammatical symbols in languages other than English. Since 1968, there have been many extensions to ASCII that have varying support throughout the world according to the prevalent language in each country. In the United States, an eight-bit ASCII variant is commonly supported that adds graphical symbols and some of the more common foreign language punctuation symbols. The original seven-bit ANSI standard ASCII mapping is shown in Table 5.1. The mappings below 0x20 are invisible control codes such as tab (0x09), carriage return (0x0D), and line-feed (0x0A). Some of the control codes are not in widespread use anymore.

5.4 RS-232

Aside from a common data representation format, communication signaling such as framing or error detection also requires standardization so that equipment manufactured by different companies can exchange information. When one begins discussing communications, an unstoppable journey into the sometimes mysterious world of industry standards begins. Navigating these standards can be tricky because of subtle differences in terminology between related standards and the everyday jargon to which the engineering community has grown accustomed. Standards are living documents that are periodically updated, revised, or replaced. This shifting base of documentation can add other challenges to fully complying with a standard.

One of the most ubiquitous serial communications schemes in use is defined by the *RS-232* family of standards. Most UARTs are designed specifically to support RS-232. Standards purists may balk at the common reference to RS-232 in the modern context, for several reasons. First, the original RS-232 document has long since been superseded by multiple revisions. Second, its name was changed first to EIA-232, then to EIA/TIA-232. And third, RS-232 is but one of a set of related standards that address asynchronous serial communications. These standards have been developed under the auspices of the Electronics Industry Alliance (formerly the Electronics Industry Association) and Telecommunications Industry Association. Technically, EIA/TIA-232 (first introduced in 1962 as RS-232) standardizes the 25-pin D-subminiature (DB25) connector and pin assignment along with an obsolete electrical specification that had limited range. EIA/TIA-423 standardizes the modern electrical characteristics that enable communication at speeds up to 100 kbps over short distances (10 m). EIA/TIA-574 standardizes the popular nine-pin DE9 connector that is used on most new "RS-232" equipped devices. These days, when most people talk about an RS-232 port, they are referring to the overall RS-232 family of related serial interfaces. In fairness to standards purists, this loose terminology is partially responsible for confusion among those who implement and use RS-232. From a practical perspective, however, it is most common to use the term RS-232 with additional qualifiers (e.g., 9-pin or 25-pin) to convey your point. In fact, if you start mentioning EIA/TIA-574 and 423, you will probably be met by blank stares from most engineers. This somewhat shady practice is continued here because of its widespread acceptance in industry.

RS-232 specifies that the least-significant bit of a byte is transmitted first and is framed by a single start bit and one or two stop bits. Common RS-232 data rates are known to many computer users.

Standard bit rates are 2^N multiples of 300 bps. In the 1970s, 300 bps serial links were common. During the 1980s, links went from 1,200 to 2,400, to 9,600 bps. RS-232 data links now operate at speeds from 19.2 to 153.6 kbps. Standard RS-232 bit rates are typically divided down from reference clocks such as 1.843, 3.6864, 6.144, and 11.0592 MHz. This explains why many microprocessors operate at oddball frequencies instead of even speeds such as 5, 10, or 12 MHz.

RS-232 defines signals from two different perspectives: *data communications equipment* (DCE) and *data terminal equipment* (DTE). DCE/DTE terminology evolved in the early days of computing when the common configuration was to have a dumb terminal attached to a modem of some sort to enable communication with a mainframe computer in the next room or building. A person would sit at the DTE and communicate via the DCE. Therefore, in the early 1960s, it made perfect sense to create a communication standard that specifically addressed this common configuration. By defining a set of DTE and DCE signals, not only could terminal and modem engineers design compatible systems, but cabling would be very simple: just wire each DTE signal straight through to each DCE signal. To further reduce confusion, the DTE was specified as a male DB-25 and the DCE as a female DB-25. Aside from transmit and receive data, hardware handshaking signals distinguish DCE from DTE. Some signals are specific to modems such as *carrier detect* and *ring indicator* and are still used today in many modem applications.

The principle behind RS-232 hardware handshaking is fairly simple: the DTE and DCE indicate their operational status and ability to accept data. The four main handshaking signals are *request to send* (RTS), *clear to send* (CTS), *data terminal ready* (DTR), and *data set ready* (DSR). DTR/DSR enable the DTE and DCE to signal that they are both operational. The DTE asserts DTR, which is sensed by the DCE and vice versa with DSR. RTS/CTS enable actual data transfer. RTS is asserted by the DTE to signal that the DCE can send it data. CTS is asserted by the DCE to signal the DTE that it can send data. In the case of a modem, carrier detect is asserted to signal an active connection, and ring indicator is asserted when the telephone line rings, signaling that the DTE can instruct the modem to answer the phone.

In a *null-modem*

TABLE 5.2 RS-232 DTE Pin Assignments

DB25 DTE	DE9 DTE	Signal	Direction: DTE/DCE	Description
1	–	Shield	⇔	Shield/chassis ground
2	3	TXD	⇒	Transmit data
3	2	RXD	⇐	Receive data
4	7	RTS	⇒	Request to send
5	8	CTS	⇐	Clear to send
6	6	DSR	⇐	Data set ready
7	5	Ground	⇔	Signal ground
8	1	DCD	⇐	Data carrier detect
9	–	+V	⇔	Power
10	–	–V	⇔	Power return
11	–			Unused
12	–	SCF	⇐	Secondary line detect
13	–	SCB	⇐	Secondary CTS
14	–	SBA	⇒	Secondary TXD
15	–	DB	⇐	DCE element timing
16	–	SBB	⇐	Secondary RXD
17	–	DD	⇐	Receiver element timing
18	–			Unused/local-loopback
19	–	SCA	⇒	Secondary RTS
20	4	DTR	⇒	Data terminal ready
21	–	CQ	⇐	Signal quality detect
22	9	RI	⇐	Ring indicator
23	–	CH/CI	⇔	Data rate detect
24	–	DA	⇒	Transmitter element timing
25	–			Unused/test-mode

boxes can be purchased that consist of jumper wires, switches, and LEDs to help troubleshoot RS-232 connectivity problems by reconfiguring interfaces on the fly as the LEDs indicate which signals are active at any given moment. As a result of the male/female gender differences of various DB25/DE9 connectors, there are often cabling problems for which one needs to connect two males or two females together. Once again, the industry has responded by providing a broad array of gender-matching cables and adapters. On a conceptual level, these problems are simple; in practice, the per-

The basic practical result of transmission line theory is that, as the speed-distance product of an electrical signal increases, the signal tends to reflect off the ends of wires and bounce back and forth on the wire. When slow signals travel relatively moderate distances, the speed-distance product is not large enough to cause this phenomenon to any noticeable degree. Fast signals traveling over very short distances may also be largely immune to such reflections. However, when RS-422 signals travel over several kilometers, the speed-bandwidth product is great enough to cause previously transmitted data signals to reflect and interfere with subsequent data. This problem can be largely solved by properly *terminating* the receiving end of the transmission line with the line's *characteristic impedance*, Z_O. Typical coaxial and twisted-pair transmission lines have Z_O = 50, 75, or 110 Ω. Briefly put, Z light. If an acoustic medium such as a telephone is used to send the data, the carrier is audible in the range of several kilohertz.

Frequency shift keying (FSK), a type of *frequency modulation* (FM) is a scheme that can be used to transmit multiple bits simultaneously without resorting to multiple levels of amplitude by using AM. FSK represents multiple bits by varying the frequency rather than the amplitude of the carrier. This constant amplitude approach is less susceptible to noise. Figure 5.10 shows 4-FSK modulation, in which each of the four frequency steps represents a different two-digit binary value.

A general term for a modulated data unit is a *baud*. If 2-ASK is used, each baud corresponds to one bit. Therefore, the baud rate matches the bit rate. However, the 4-FSK example shows that each baud represents two bits, making the bit rate twice that of the baud rate. This illustrates that baud rate and bit rate are related but not synonymous, despite common misuse in everyday conversation. Engineers who design modulation circuitry care about the baud rate, because it specifies how many unique data units can be transmitted each second. They also try to squeeze as many bits per baud as possible to maximize the overall bit rate of the modulator. Engineers who use modulators as black-box components do not necessarily care about the baud rate; rather, it is the system's bit rate that matters to the end application.

Enter the *modem*. A modem is simply a device that incorporates a modulator and demodulator for a particular transmission medium. The most common everyday meaning of modem is one that enables a computer to transfer bits over an analog telephone line. These modems operate using different modulation schemes depending on their bit rate. Early 300- and 1,200-bps modems operate using FSK and *phase shift keying* (PSK). Later modems, including today's 33.6- and 56-kbps models, operate using variations of

these types of networks do not observe all traffic that traverses the network; rather, the network itself contains some intelligence when it comes to delivering a packet. A node in a logically starred netacross spans of twisted-pair wire exceeding 1.2 km. Unlike RS-422, the RS-485 standard allows up to 32 transmit/receive nodes on a single twisted pair that is terminated at each end as shown in Fig. 5.14. Modern low-load receivers that draw very little current from the RS-485 bus can be used to increase the number of nodes on an RS-485 network well beyond the original 32-node limit to 256 nodes or more. A single pair of wires is used for both transmit and receive, meaning that the system is capable of *half-duplex* (one-way) operation rather than *full-duplex* operation (both directions at the same time). Half-duplex operation restricts the network to one-way exchange of information at any given time. When node A is sending a packet to node B, node B cannot simultaneously send a packet to node A.

RS-485 directly supports the implementation of bus networks. Bus topologies are easy to work with, because nodes can directly communicate with each other without having to pass through other nodes or semi-intelligent hubs. However, a bus network requires provisions for sharing access to be built into the network protocol. In a centralized arbitration scheme, a master node gives permission for any other node to transmit data. This permission can be a request-reply scheme whereby slave nodes do not respond unless a request for data is issued. Alternatively, slave nodes can be periodically queried by the master for transmit requests, and the master can grant permissions on an individual-node basis. There are many centralized arbitration schemes that have been worked out over the years.

A common distributed arbitration scheme on a bus network is *collision detection* with random back-off. When a node wants to transmit data, it first waits until the bus becomes idle. Once idle, the node begins transmitting data. However, when the node begins transmitting, there is a chance that one or more nodes have been waiting for an opportunity to begin transmitting and that they will begin transmitting at the same time. Collision detection circuits at each node determine that more than one node is transmitting, and this causes all active transmitters to stop. Figure 5.15 shows the imple-

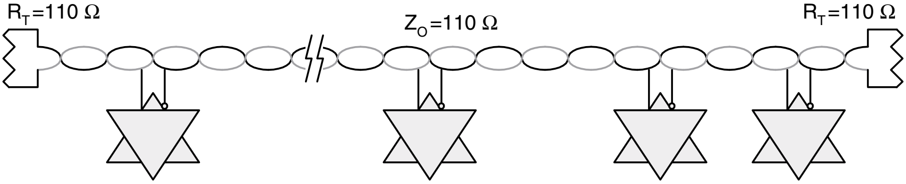


FIGURE 5.14 RS-485 bus topology.

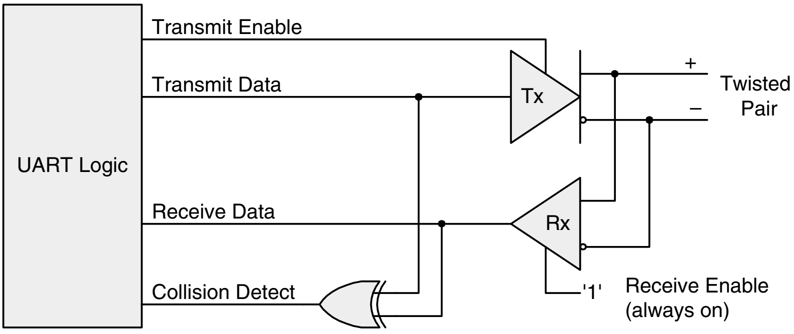


FIGURE 5.15 RS-485 collision detection transceiver.

mentation of an RS-485 transceiver with external collision detection logic. A transmit enable signal exists to turn off the transmitter when the UART is not actively sending data. Unlike an RS-422 transmitter that does not have to share access with others, the RS-485 transmitter must turn itself off when not sending data to enable others to transmit.

When transmitting, the receiver returns the logical state of the twisted-pair bus. If the bus is not at the same state as the transmitted data, a collision is most likely being caused by another transmitter trying to drive the opposite logic state. An XOR gate implements this collection detect, and the XOR output must be sampled only after allowing adequate time for the bus to settle to a stable state following the assertion of each bit from the transmitter.

Once a collision has been detected by each node and the transmitters are disabled, each node waits a different length of time before retransmitting. If all delays were equal, multiple nodes would get caught in a deadlock situation wherein each node keeps trying to transmit after the same delay interval. Random back-off delays are pseudo-random so as to not unfairly burden some nodes with consistently longer delays than other nodes. At the end of the delay, one of the nodes begins transmitting first and gains control of the bus by default. The other waiting nodes eventually exit from their delays and observe that the bus is already busy, indicating that they must wait their turn until the current packet has been completed. If, by coincidence, another node begins transmitting at the same time that the first node begins, the back-off process begins again. It is statistically possible for this process to occur several times in a row, although the probability of this being a frequent event is small in a properly designed network. A bus network constructed with too many nodes trying to send too much data at the same time can exhibit very poor performance, because it would be quite prone to collisions. In such a case, the solution may be to either reduce the network traffic or increase the network's bandwidth.

5.10 A SIMPLE RS-485 NETWORK

An example of a simple but effective network implemented with RS-485 serves as a vehicle to discuss how packet formats, protocols, and hardware converge to yield a useful communications medium. The motivation to create a custom RS-485 network often arises from a need to deploy remote actuators and data-acquisition modules in a factory or campus setting. A central computer may be located in a factory office, and it may need to periodically gather process information (e.g., temperature, pressure, fluid-flow rate) from a group of machines. Alternatively, a security control console located in one building may need to send security camera positioning commands to locations throughout the campus. Such applications may involve a collection of fairly simple and inexpensive microprocessor-based modules that contain RS-485 transceivers. Depending on the exact physical layout, it may or may not be practical to wire all remote nodes together in a single twisted-pair bus. If not, a logical bus can be formed by creating a hybrid star/bus topology as shown in Fig. 5.16. A central hub electrically connects the individual star segments so that they function electrically as a large bus but do not require a single wire to be run throughout the entire campus.

As shown, the hub does not contain any intelligent components—it is a glorified junction box. This setup is adequate if the total length of all star segments does not exceed 1.2 km, which is within the electrical limitations of the RS-485 standard. While simple, this setup suffers from a lack of fault tolerance. If one segment of the star wiring is damaged, the entire network may cease operation because, electrically, it is a single long pair of wires. Both the distance and fault-tolerance limitations can be overcome by implementing an active hub that contains *repeaters* on each star segment and smart switching logic to detect and isolate a broken segment. A repeater is an active two-port device that amplifies or regenerates the data received on one port and transmits it on the other port. An RS-

A single parity bit cannot guarantee the detection of multiple errors in the same byte, because such errors can mask themselves. For example, two bit errors can flip a data bit and the parity bit itself, making it impossible for the receiver to detect the error. More complex error detection schemes are available and are more difficult to fool. Although no error detection solution is perfect, some schemes reduce the probability of undetected errors to nearly zero.

If a packet is received with an error, it cannot be acted upon normally, because its contents are suspect. For the purposes of devising a useful error-handling scheme, packet errors can be divided into two categories: those that corrupt the destination/source address information and those that do not. Parity errors that corrupt the packet's addresses must result in the packet being completely ignored, because the receiving node is unable to generate a reply message to the originator indicating that the packet was corrupted. If the source address is corrupted, the receiver does not know to whom to reply. If the destination address is corrupted, the receiver does not know whether it is the indented recipient.

In the case of an address error in which the received packet is ignored, the originator must implement some mechanism to recover from the packet loss rather than waiting indefinitely for a reply that will never arrive. A reply *timeout* can be implemented by an originator each time a packet is sent that requires a corresponding reply. A timeout is an arbitrary delay during which an originating node waits before giving up on a response from a remote node. Timeouts are common in networks because, if a packet is lost due to an error, the originator should not wait indefinitely for a response that will never come. Establishing a timeout value is a compromise between not giving up too quickly and missing a slower-than-normal reply and waiting too long and introducing unacceptable delays in system functionality when a packet is lost. Depending on the time it takes to send a packet on a network and the nodes' typical response time, timeouts can range from microseconds to minutes. Typical timeouts are often expressed in milliseconds.

When an originator times-out and concludes that its requested data somehow got lost, it can resend the request. If, for example, a security control node sends a request for a camera to pan across a

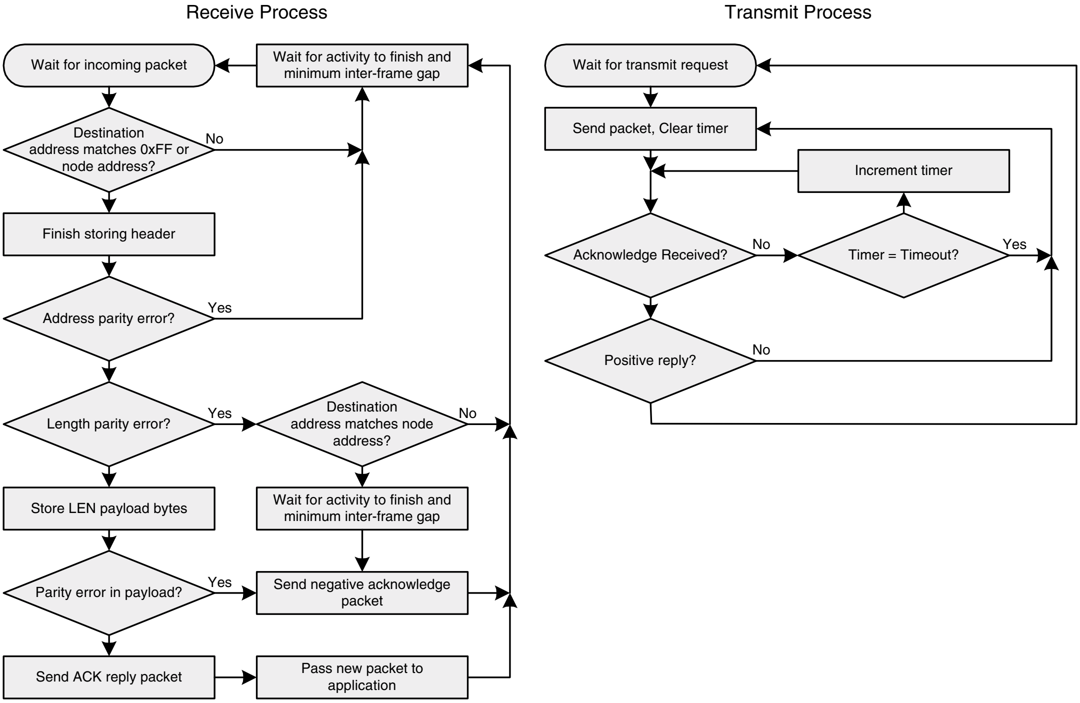


FIGURE 5.17 Hypothetical network driver flowchart.

to traditional parallel buses. Computer architectures often include a variety of microprocessor peripheral devices with differing bandwidth requirements. Main memory, both RAM and ROM, is a central part of computer architecture and is a relatively high-bandwidth element. The fact that the CPU must continually access main memory requires a simple, high-bandwidth interface—a parallel bus directly or indirectly driven by the CPU. Other devices may not be accessed as often as main memory and therefore have a substantially lower bandwidth requirement. Peripherals such as data acquisition ICs (e.g., temperature sensors), serial number EEPROMs, or liquid crystal display (LCD) controllers might be accessed only several times each second instead of millions of times per second. These peripherals can be directly mapped into the CPU's address space and occupy a spot on its parallel bus, but as the number of these low-bandwidth peripherals increases, the complexity of attaching so many devices increases.

Short-distance serial data links can reduce the cost and complexity of a computer system by reducing interchip wiring, minimizing address decoding logic, and saving pins on IC packages. In such a system, the CPU is connected to a serial controller via its parallel bus, and most other peripherals are connected to the controller via several wires in a bus topology as shown in Fig. 5.18.

Such peripherals must be specifically designed with serial interfaces, and many are. It is common for low-bandwidth peripheral ICs to be designed in both parallel and serial variants. In fact, some devices are manufactured with only serial interfaces, because their economics overwhelmingly favors the reduction in logic, wiring, and pins of a serial data link. A temperature sensor with a serial interface can be manufactured with just one or two signal pins plus power. That same sensor might

CHAPTER 6

Instructive Microprocessors and Microcomputer Elements

Microprocessors, the heart of digital computers, have been in a constant state of evolution since Intel developed the first general-purpose microprocessors in the early 1970s. Intel's four-bit 4004 made ist today. The 8080 was housed in a 40-pin DIP, featured a 16-bit address bus and an 8-bit data bus, and ran at 2 MHz. It also implemented a conventional stack pointer that enabled deep stacks in external memory (Intel's earlier microprocessors had internal stacks with very limited depth). The 8080 became extremely popular as a result of its performance and rich, modern instruction set. This popularity was evidenced two years later, in 1976, with Intel's enhanced 8085 and competitor Zilog's famous Z80. Designed by former Intel engineers, the Z80 was based heavily on the 8080 to the point of having a partially compatible instruction set.

Both the 8085 and Z80 were extremely popular in a variety of computing platforms from hobbyists to mainstream commercial products to video arcade games. The 8085 architecture influenced the famous 16-bit 8086 family whose strong influence continues to this day in desktop PCs. The Z80 eventually lost the mainstream microprocessor war and migrated to microcontrollers that are still available for new designs from Zilog.

As microprocessors progress, technologies that used to be leading edge first become mainstream and then appear quite pedestrian. Along the way, some microprocessor families branch into multiple product lines to suit a variety of target applications. The high-end computing market gets most of the publicity and accounts for the major technology improvements over time. Lower-end microprocessors are either made obsolete after some time or find their way into the *embedded* market. Embedded microprocessors and systems are those that may not appear to the end user as a computer, or they may not be visible at all. Instead, embedded microprocessors typically serve a control function in a machine or another piece of equipment. This is in contrast to the traditional computer with a keyboard and monitor that is clearly identified as a general-purpose computer.

Integrated microprocessor products are called *microcontrollers*, a term that has already been introduced. A microcontroller is a microprocessor integrated with a varying mix of memory and peripherals on a single chip. Microcontrollers are almost always found in embedded systems. As with many industry terms, *microcontrollers* can mean very different things to different people. In general, a microcontroller contains a relatively inexpensive microprocessor core with a complement of onboard peripherals that enable a very compact, yet complete, computing system—either on a single chip or relatively few chips. There is a vast array of single-chip microcontrollers on the market that integrate quantities of both RAM and ROM on the same chip along with basic peripherals including serial communications controllers, timers, and general I/O signal pins for controlling LEDs, relays, and so on. Some of the smallest microcontrollers can cost less than a dollar and are available in packages with as few as eight pins. Such devices can literally squeeze a complete computer into the area of a fingernail. More complex microcontrollers can cost tens of dollars and provide external microprocessor buses for memory and I/O expansion. At the very high end, there are microcontrollers available for well over $100 that include 32-bit microprocessors running at hundreds of megahertz, with integrated Ethernet controllers and DMA. Manufacturers typically refer to these high-end microcontrollers with unique, proprietary names to differentiate them from the aforementioned class of inexpensive devices.

6.2 MOTOROLA 6800 EIGHT-BIT MICROPROCESSOR FAMILY

As the microprocessor market began to take off, Motorola jumped into the fray and introduced its eight-bit 6800 in 1974, shortly after the 8080 first appeared. While no longer available as a discrete microprocessor, the 6800 is significant, because it remains in Motorola's successful 68HC05/68HC08 and 68HC11 microcontroller families and also serves as a vehicle with which to learn the basics of computer architecture. Like the 8080, the 6800 is housed in a 40-pin DIP and features a 16-bit address bus and an 8-bit data bus. All of the basic register types of a modern microprocessor are implemented in the 6800, as shown in Fig. 6.1: a program counter (PC), stack pointer (SP), index register (X), two general-purpose accumulators (ACCA and ACCB), and status flags set by the ALU in the condition code register (CCR). ACCA is the primary accumulator, and some instructions operate only on this register and not ACCB. A half-carry flag is included to enable efficient binary coded decimal (BCD) operations. After adding two BCD values with normal binary arithmetic, the half-carry is used to convert illegal results back to BCD. The 6800 provides a special instruction, decimal adjust ACCA (DAA), for this specific purpose. A somewhat out-of-place interrupt mask bit is also implemented in the CCR, because this was an architecturally convenient place to locate it. Bits in the CCR are modified through either ALU operations or directly by transferring the value in ACCA to the CCR.

The 6800 supports three interrupts: one nonmaskable, one maskable, and one software interrupt. More recent variants of the 6800 support additional interrupt sources. A software interrupt can be used by any program running on the microprocessor to immediately jump to some type of maintenance routine whose address does not have to be known by the calling program. When the software interrupt instruction is executed, the 6800 reads the appropriate interrupt vector from memory and jumps to the indicated address. The 6800's reset and interrupt vectors are located at the top of memory, as listed in Table 6.1, which generally dictates that the boot ROM be located there as well. For example, an 8-kB 27C64 EPROM (8,192 bytes = 0x2000 bytes) would occupy the address range 0xE000 through 0xFFFF. Each vector is 16 bits wide, enough to specify the full address of the associated routine. The MSB of the address, A[15:8], is located in the low, or even, byte address, and the LSB, A[7:0] is located in the high, or odd, byte address.

TABLE 6.1 6800 Reset and Interrupt Vectors

Vector Address	Purpose
0xFFFE/0xFFFF	Reset
0xFFFC/0xFFFD	Nonmaskable interrupt
0xFFFA/0xFFFB	Software interrupt
0xFFF8/0xFFF9	Maskable interrupt

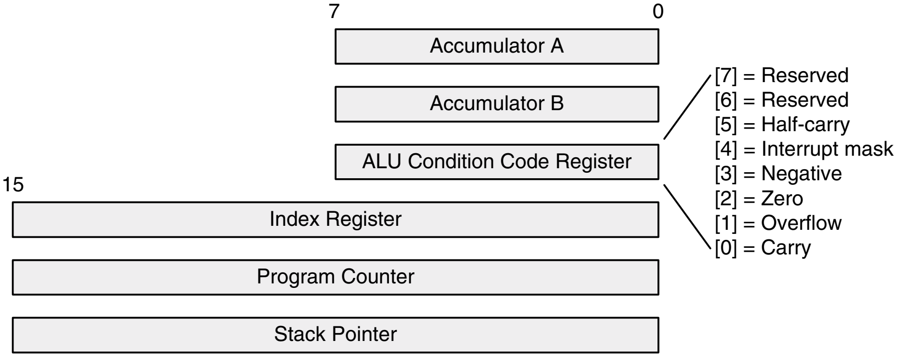


FIGURE 6.1 6800 registers.

An external clock driver circuit that provides a two-phase clock (two clock signals 180° out of phase with respect to each other) is required for the original 6800. Motorola simplified the design of 6800-based computer systems by introducing two variants, the 6802 and 6808. The 6802 includes an on-board clock driver circuit of the type that is now standard on many microprocessors available today. Such clock drivers require only an external *crystal* to create a stable, reliable oscillator with which to clock the microprocessor. A crystal is a two-leaded component that contains a specially cut quartz crystal. The quartz can be made to resonate at its natural frequency by electrical stimulus created within the microprocessor's on-board clock driver circuitry. A crystal is necessary for this purpose, because its oscillation frequency is predictable and stable. The 6802 also includes 128 bytes of on-board RAM to further simplify certain systems that have small volatile memory requirements. For customers who wanted the simplified clocking scheme of the 6802 without paying for the on-board RAM, Motorola's 6808 kept the clocking and removed the RAM.

Using a 6802 with its internal RAM, a functional computer could be constructed with only two chips: the 6802 and an EPROM. Unfortunately, such a computer would not be very useful, because it would have no I/O with which to interact with the outside world. Motorola manufactured a variety of peripheral chips intended for direct connection to the 6800 bus. Among these were the 6821 peripheral interface adapter (PIA) and the 6850 asynchronous communications interface adapter (ACIA), a type of UART. The PIA provides 20 I/O signals arranged as two 8-bit parallel ports, each with two control signals. Applications including basic pushbutton sensing and LED driving are easy with the 6821. The 6800 bus uses asynchronous control signals, meaning that memory and I/O devices do not explicitly require access to the microprocessor clock to communicate on the bus. However, many of the 6800 peripherals require their own copy of the clock to run internal logic.

As with all synchronous logic, the 6800's bus is internally controlled by the microprocessor clock, but the nature of the control signals enables asynchronous read and write transactions without referencing that clock, as shown in Fig. 6.2. An address is placed onto the bus along with the proper state of the R/W select signal (read = 1, write = 0) and a valid memory address (VMA) enable that indicates an active bus cycle. In the case of a write, the write data is driven out some time later. For reads, the data must be returned fast enough to meet the microprocessor's timing specifications. The 6802/6808 were manufactured in 1-, 1.5-, and 2-MHz speed grades. At 2 MHz, a peripheral device has to respond to a read request with valid data within 210 ns after the assertion of address, R/W, and VMA. A peripheral has up to 290 ns from the assertion of these signals to complete a write transaction.* In a real system, VMA, combined with address decoding logic, would drive the individual chip select signals to each peripheral.

In some situations, slow peripherals may be used that cannot execute a bus transaction in the time allowed by the microprocessor. The 6800 architecture deals with this by stretching the clock during

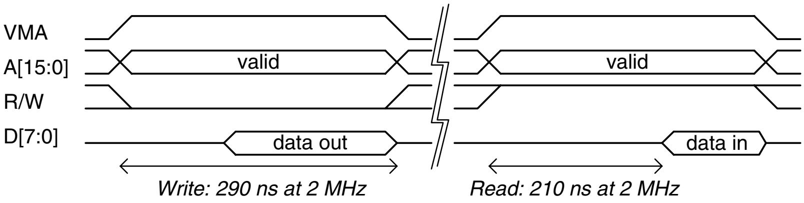


FIGURE 6.2 6802/6808 basic bus timing.

* *8-Bit Microprocessor and Peripheral Data,* Motorola, 1983, pp. 3–182.

IBM PC because of its simplicity and low cost. The 8048 was manufactured in a 40-pin DIP and could be expanded with external memory and peripherals via an optional external address/data bus. However, when operated as a nonexpanded single-chip computer, the pins that would otherwise function as its bus were available for general I/O purposes—a practice that is fairly standard on microcontrollers.

Motivated by the popularity of the 8048, Intel introduced the 8051 microcontroller in 1980, which is substantially more powerful and flexible. The 8051's basic architecture is shown in Fig. 6.3. It contains 128 bytes of RAM, 4 kB of ROM, two 16-bit timer/counters, and a serial port. Registers within the microprocessor are 8 bits wide except for the 16-bit data pointer (DPTR) and program counter (PC). Memory is divided into mutually exclusive program and data sections that each can be expanded up to 64 kB in size via an external bus. Expansion is accomplished by borrowing pins from two of the four 8-bit I/O ports. Intel manufactured several variants of the 8051. The 8052 doubled the amount of on-chip memory to 256 bytes of RAM and 8 kB of ROM and added a third timer. The 8031/8032 are 8051/8052 chips without on-board ROM. The 8751/8752 are 8051/8052 devices with EPROM instead of mask ROM. As time went by and the popularity of the 8051 family increased, other companies licensed the core architecture and developed many variants with differing mixes of memory and peripherals.

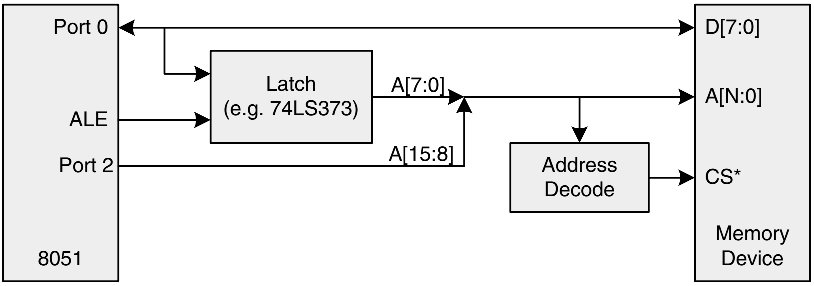


FIGURE 6.4 8051 system with external address latch.

for valuable I/O functions. Some applications may suffice with just an eight-bit external address bus. For example, if the only expansion necessary were a special purpose I/O device, 256 bytes would probably be more than enough to communicate with the device. However, some applications demand a fully functional 16-bit external address bus. In these situations, port 2 is used to drive the upper address bits, A[15:8].

The 8051's microprocessor is very capable for such an early microcontroller. It includes integer multiply and divide instructions that utilize eight-bit operands in the accumulator and B register, and it then places the result back into those registers. The stack, which grows upward in memory, is restricted to on-board RAM only (256 bytes at most), so only an eight-bit stack pointer is implemented. Aside from the general-purpose accumulator and B registers, the 8051 instruction set can directly reference 8 byte-wide general-purpose registers, numbered R0 through R7, that are mapped as 4 banks in the lower 32 bytes of on-board RAM. The active register bank can be changed at any time by modifying two bank-select bits in the status word. The map of on-board data memory is shown in Table 6.2. At reset, register bank 0 is selected, and the stack pointer is set to 0x07, meaning that the stack will actually begin at location 0x08 when the first byte is eventually pushed. Above the register banks is a 16-byte (128-bit) region of memory that is bit addressable. Microcontroller applications often involve reading status information, checking certain bits to detect particular events, and then triggering other events. Using single bits rather than whole bytes to store status information saves precious memory in a microcontroller. Therefore, the 8051's bit manipulation instructions can make efficient use of the chip's resources from both instruction execution and memory usage perspectives. The remainder of the lower 128-byte memory region contains 80 bytes of general-purpose memory.

The upper 128 bytes of data memory are split into two sections: special-function registers and RAM. Special-function registers are present in all 8051 variants, but their definitions change according to the specific mix of peripherals in each variant. Some special-function registers are standard across all 8051 variants. These registers are typically those that were implemented on the original 8051/8052 devices and include the accumulator and B registers; the stack pointer; the data pointer; and serial port, timer, and I/O port control registers. Each time a manufacturer adds an on-board peripheral to the 8051, accompanying control registers are added into the special-function memory region.

On variants that incorporate 256 bytes of on-board RAM, the upper 128 bytes are also mapped into a parallel region alongside the special-function registers. Access between RAM and special-function registers is controlled by the addressing mode used in a given instruction. Special-function registers are accessed with direct addressing only. Therefore, such an instruction must follow the opcode with an eight-bit address. The upper 128 bytes of RAM are accessed with indirect addressing only. Therefore, such an instruction must reference one of the eight general-purpose registers (R0

Although the specific timing delays of program memory and data memory reads are different, they exhibit the same basic sequence of events. (More time is allowed for data reads than for instruction reads from program memory.) Therefore, if the engineer properly accounts for the timing variations by selecting memory and logic components that are fast enough to satisfy the PSEN* and RD* timing specifications simultaneously, program and data memory can actually be merged into a unified memory space external to the chip. Such unification can be performed by generating a general memory read enable, MRE*, that is the AND function of PSEN* and RD*. In doing so, whenever either read enable is driven low by the 8051, MRE* will be low. This can benefit some applications by turning the 8051 into a more general-purpose computing device that can load a program into its "data memory" and then execute that same program from "program memory." It also enables indexed addressing to operate on data memory, which normally is restricted to indirect addressing as discussed previously.

Timers such as those found in the 8051 are useful for either counting external events or triggering low-frequency events themselves. Each timer can be configured in two respects: whether it is a timer or counter, and how the count logic functions. The selection of timer versus counter is a decision between incrementing the count logic based on the microcontroller's operating frequency or on an excounters, as a single 8-bit counter with a 5-bit prescaler, and as a single 8-bit counter with an 8-bit the expense of the ceramic DIP in which they are most often found. More specialized 8051 variants may be available only through manufacturers' authorized distributors.

6.4 MICROCHIP PIC® MICROCONTROLLER FAMILY

By the late 1980s, microcontrollers and certain microprocessors were well established in embedded control applications. Despite advances in technology, not many devices could simultaneously address the needs for low power, moderate processing throughput, very small packages, and diverse integrated peripherals. Microchip Technology began offering a family of small peripheral interface controller (PIC®)* devices in the early 1990s that addressed all four of these needs. Microchip developed the compact PIC architecture based on a *reduced instruction set core* (RISC) microprocessor. The chips commonly run at up to 20 MHz and execute one instruction every machine cycle (four clock cycles)—except branches that consume two cycles. The key concept behind the PIC family is simplicity. The original 16C5x family, shown in Fig. 6.7, implements a 33-instruction microprocessor core with a single working register (accumulator), W, and only a two-entry subroutine stack. These devices contain as little as 25 bytes of RAM and 512 bytes of ROM, and some are housed in an 18-pin package that can be smaller than a fingernail. The PIC devices are not expandable via an external bus, further saving logic. This minimal architecture is what enables relatively high performance processing with low power consumption in a tiny package. Low-power operation is also coupled with a wide operating voltage range (2 to 6.25 V), further simplifying certain systems by not always requiring voltage regulation circuits.

No interrupt feature is included, which is a common criticism of the architecture; this was fixed in subsequent PIC microcontroller variants. PIC devices are, in general, fully static, meaning that they can operate at an arbitrarily low frequency; 32 kHz is sometimes used in very power-sensitive appli-

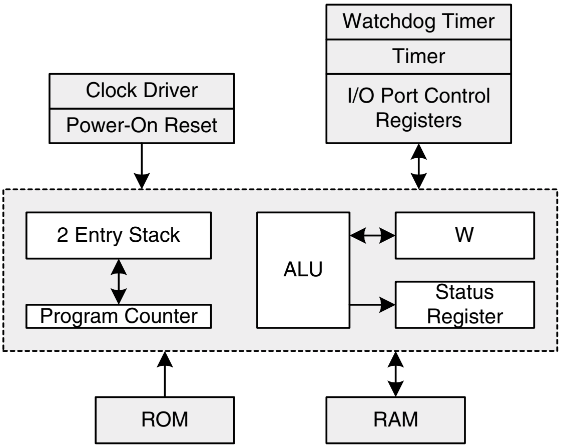


FIGURE 6.7 PIC microcontroller 16C5x architecture.

[*] The Microchip name, PIC, and PICmicro are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries.

Microchip extended the 16C5x's architecture and features with the 16C6x and 16C7x families. The 16C5x's advantages of low power consumption, wide operating voltage range, and small size are retained. Improvements include more memory (up to 368 bytes of RAM and 8 kB of ROM), a more versatile microprocessor core with interrupt capability, an eight-level stack, and a wider selection of on-board peripherals including additional timers, serial ports, and *analog-to-digital* (A/D) converters. (An A/D converter is a circuit that converts an analog voltage range into a range of binary values. An 8-bit A/D converter covering the range of 0 to 5 V would express voltages in that range as a byte value with a resolution of 5 V ÷ (2^8 – 1) increments = 19.6 mV per increment.) Between four and eight A/D converters are available in 'C7x devices.

Some PIC microcontrollers contain two serial ports on the same chip: an asynchronous port suitable for RS-232 applications and a synchronous port capable of SPI or I^{2}C operation in conjunction with other similarly equipped ICs in a system. At the other end of the spectrum, very small PIC devices are available in eight-pin packages—small enough to fit almost anywhere.

6.5 INTEL 8086 16-BIT MICROPROCESSOR FAMILY

Intel moved up to a 16-bit microprocessor, the 8086, in 1978—just two years after introducing the 8085 as an enhancement to the 8080. The "x86" family is famous for being chosen by IBM for their original PC. As PCs developed during the past 20 years, the x86 family grew with the industry—first to 32 bits (80386, Pentium) and more recently to 64 bits (Itanium). While the 8086 was a new architecture, it retained certain architectural characteristics of the 8080/8085 such that assembly language programs written for its predecessors could be converted over to the 8086 with little or no modification. This is one of the key reasons for its initial success.

The 8086 contains various 16-bit registers as shown in Fig. 6.9, some of which can be manipulated one byte at a time. AX, BX, CX, and DX are general-purpose registers that have alternate functions and that can be treated as single 16-bit registers or as individual 8-bit registers. The accumulator, AX, and the flags register serve their familiar functions. BX can serve as a general pointer. CX is a loop iteration count register that is used inherently by certain instructions. DX is used as a companion register to AX when performing certain arithmetic operations such as integer division or handling long integers (32 bits).

The remaining registers are pointers of various tb8wndEDinfasilU—Hfgia x9HHflf x9Hw8Aonta9AUfo—

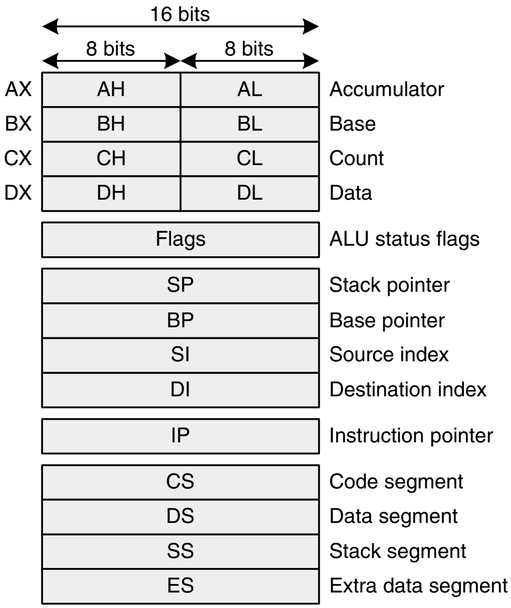
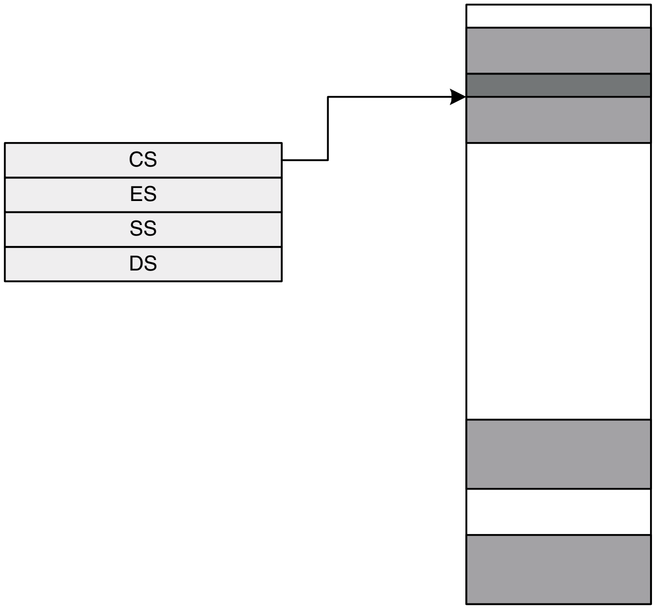


FIGURE 6.9 8086 register set.



Inside the microprocessor, this math is performed by shifting the segment pointer (0x135F) left by four bits and then adding the offset pointer (0x0102) as shown below.

	1	3	5	F	0	Segment pointer
+		0	1	0	2	Offset pointer
	1	3	6	F	2	Effective address

This segmented addressing scheme has some awkward characteristics. First, programs must organize their instructions and data into 64-kB chunks and properly keep track of which portions are being accessed. If data outside of the current segments is desired, the appropriate segment register must be updated. Second, the same memory location can be represented by multiple combinations of segment and offset values, which can cause confusion in sorting out which instruction is accessing which location in memory. Nonetheless, programmers and the manufacturers of their development tools have figured out ways to avoid these traps and others like them.

Instructions that reference memory implicitly or explicitly determine which offset pointer is added to which segment register to yield the desired effective address. For example, a push or pop instruction inherently uses the stack pointer in combination with the stack segment register. However, an instruction to move data from memory to the accumulator can use one of multiple pointer registers relative to any of the segment registers.

The 8086's reset and interrupt vectors are located at opposite ends of the memory space. On reset, the instruction pointer is set to 0xFFFF0, and the microprocessor begins executing instructions from this address. Therefore, rather than being a true vector, the 16-byte reset region contains normal executable instructions. The interrupt vectors are located at the bottom of the memory space starting from address 0, and there are 256 vectors, one for each of the 256 interrupt types. Each interrupt vector is composed of a 2-byte segment address and a 2-byte offset address, from which a 20-bit effective address is calculated. When the 8086's INTR pin is driven high, an interrupt acknowledge process begins via the INTA* output pin. The 8086 pulses INTA* low twice and, on the second pulse, the interrupting peripheral drives an interrupt type, or vector number, onto the eight lower bits of the data bus. The vector number is used to index into the interrupt vector table by multiplying it by 4 (shifting left by two bits), because each vector consists of four bytes. For example, interrupt type 0x03 would cause the microprocessor to fetch four bytes from addresses 0x0C through 0x0F. Interrupts triggered by the INTR pin are all maskable via an internal control bit. Software can also trigger interrupts of various types via the `INT` instruction. A nonmaskable interrupt can be triggered by external hardware via the NMI pin. NMI initiates the type-2 interrupt service routine at the address indicated by the vector at 0x08-0x0B.

Locating the reset boot code at the top of memory and the interrupt vectors at the bottom often leads to an 8086 computer architecture with ROM at the top and some RAM at the bottom. ROM must be at the top, for obvious reasons. Placing the interrupt vector table in RAM enables a flexible system in which software applications can install their own ISRs to perform various tasks. On the original IBM PC platform, it was not uncommon for programs to insert their own ISR addresses into certain interrupt vectors located in RAM. The system timer and keyboard interrupts were common objects of this activity. Because the PC's operating system already implemented ISRs for these interrupts, the program could redirect the interrupt vector to its own ISR and then call the system's default ISR when its own ISR completed execution. If properly done, this interrupt chaining process could add new features to a PC without harming the existing housekeeping chores performed by the standard ISRs. Chaining the keyboard interrupt could enable a program that is normally dormant to pop up each time a particular key sequence is pressed.

memory as you would read and interpret it. The choice of "endianness" is rather religious and comes down to personal preference. Of course, if you are designing with a little-endian microprocessor, life will be made simpler to maintain the endianness consistently throughout the system.

At the time of the 8086's introduction, 16-bit desktop computer systems were almost unheard of and could be substantially more expensive than 8-bit systems as a result of the increased memory size

6.6 MOTOROLA 68000 16/32-BIT MICROPROCESSOR FAMILY

Motorola followed its 6800 family by leaping directly to a hybrid 16/32-bit microprocessor architecture. Introduced in 1979, the 68000 is a 16-bit microprocessor, due to its 16-bit ALU, but it contains all 32-bit registers and a linear, nonsegmented 32-bit address space. (The original 68000 did not bring out all 32 address bits as signal pins but, more importantly, there are no architectural limitations of using all 32 bits.) That the register and memory architecture is inherently 32 bits made the 68000 family easily scalable to a full 32-bit internal architecture. Motorola upgraded the 68000 family with true 32-bit devices, including the 68020, 68040, and 68060, until switching to the PowerPC architecture in the latter portion of the 1990s for new high-performance computing applications. Apple Computer used the 68000 family in their popular line of Macintosh desktop computers. Today, the 68000 family lives on primarily as a mid-level embedded-processor core product. Motorola manufacturers a variety of high-end microcontrollers that use 32-bit 68000 microprocessor cores. However, in recent years Motorola has begun migrating these products, as well as their general-purpose microprocessors, to the PowerPC architecture, reducing the number of new designs that use the 68000 family.

The 68000 inherently supports modern software *operating systems* (OSs) by recognizing two modes of operation: supervisor mode and user mode. A modern OS does not grant unlimited access to application software in using the computer's resources. Rather, the OS establishes a restricted operating environment into which a program is loaded. Depending on the specific OS, applications may not be able to access certain areas of memory or I/O devices that have been declared off limits by the OS. This can prevent a fault in one program from crashing the entire computer system. The OS *kernel*, the core low-level software that keeps the computer running properly, has special privileges that allow it unrestricted access to the computer for the purposes of establishing all of the rules and boundaries under which programs run. Hardware support for multiple privilege levels is crucial for such a scheme to prevent unauthorized programs from freely accessing restricted resources. As microprocessors developed over the last few decades, more hardware support for OS privileges was added. That the 68000 included such concepts in 1979 is a testimony to its scalable architecture.

Sixteen 32-bit general-purpose registers, one of which is a user stack pointer (USP), and an 8-bit condition code register are accessible from user mode as shown in Fig. 6.11. Additionally, a supervisor stack pointer (SSP) and eight additional status bits are accessible from supervisor mode. Computer systems do not have to implement the two modes of operation if the application does not require it. In such cases, the 68000 can be run permanently in supervisor mode to enable full access to all resources by all programs. The SSP is used for stack operations while in supervisor mode, and the USP is used for stack operations in user mode. User mode programs cannot change the USP, preventing them from relocating their stacks. Most modern operating systems are *multitasking*, meaning that they run multiple programs simultaneously. In reality, a microprocessor can only run one program at a time. A multitasking OS uses a timer to periodically interrupt the microprocessor, perhaps 20 to 100 times per second, and place it into supervisor mode. Each time supervisor mode is invoked, the kernel performs various maintenance tasks and swaps the currently running program with the next program in the list of running programs. This swap, or *context switch*, can entail substantial modifications to the microprocessor's state when it returns from the kernel timer interrupt. In the case of an original 68000 microprocessor, the kernel could change the return value of the PC, USP,

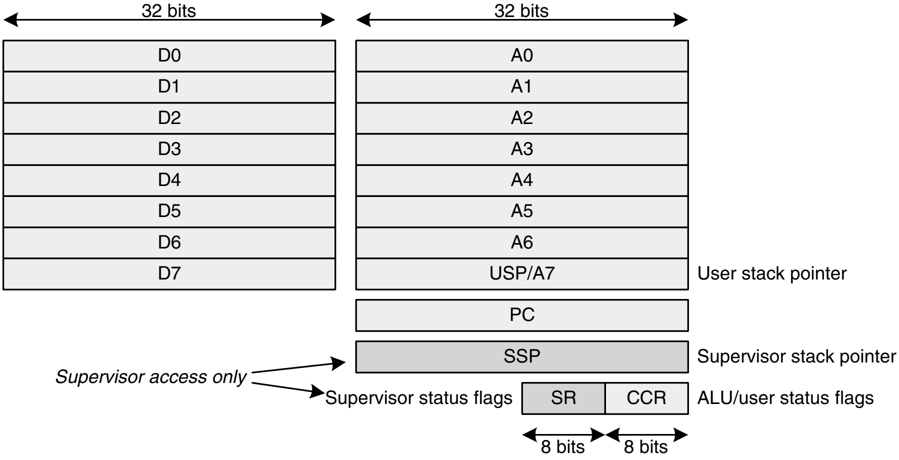


FIGURE 6.11 68000 register set.

ALU operations. All 16 registers can be used as index registers. While operating in user mode, it is illegal to access the SSP or the supervisor portion of the status register, SR. Such instructions will cause an exception, whereby a particular interrupt is asserted, which causes the 68000 to enter supervisor mode to handle the fault. (*Exception* and *interrupt* are often used synonymously in computer contexts.) Very often, the OS kernel will terminate an application that causes an exception to be generated. The registers shown above are present in all 68000 family members and, as such, are software is compatible with subsequent 68xxx microprocessors. Newer microprocessors contain additional registers that provide more advanced privilege levels and memory management. While the 68000 architecture fundamentally supports a 4-GB (32-bit) address space, early devices were limited in terms of how much physical memory could actually be addressed as a result of pin limitations in the packaging. The original 68000 was housed in a 64-pin DIP, leaving only 24 address bits usable, for a total usable memory space of 16 MB. When Motorola introduced the 68020, the first fully 32-bit 68000 microprocessor, all 32 address bits were made available. The 68000 devices are big-endian, so the MSB is stored in the lowest address of a multibyte word.

The 68000 supports a 16-MB address space, but only 23 address bits, A[23:1], are actually brought out of the chip as signal pins. A[0] is omitted and is unnecessary, because it would specify whether an even (A[0] = 0) or odd (A[0] = 1) byte is being accessed; and, because the bus is 16 bits wide, both even and odd bytes can be accessed simultaneously. However, provisions are made for byte-wide accesses in situations where the 68000 is connected to legacy eight-bit peripherals or memories. Two data strobes, upper (UDS*) and lower (LDS*), indicate which bytes are being accessed during any given bus cycle. These strobes are generated by the 68000 according to the state of the internal A0 bit and information on the size of the requested transaction. Bus transactions are triggered by the assertion of address strobe (AS*), the appropriate data strobes, and R/W* as shown in Fig. 6.12. Prior to AS*, the 68000 asserts the desired address and a three-bit function code bus, FC[2:0]. The function code bus indicates which mode the processor is in and whether the transaction is a program or data access. This information can be used by external logic to qualify transactions to certain sensitive memory spaces that may be off limits to user programs. When read data is ready, the external bus interface logic asserts data transfer acknowledge (DTACK*) to inform the microprocessor that the transaction is complete. As shown, the 68000 bus can be operated in a fully asynchronous manner. When operated asynchronously, DTACK* is removed after the strobes are

P • A • R • T • 2

ADVANCED DIGITAL SYSTEMS

This page intentionally left blank.

CHAPTER 7
Advanced Microprocessor Concepts

Computer architecture is central to the design of digital systems, because most digital systems are, at their core, computers surrounded by varying mixes of interfaces to the outside world. It is difficult to know at the outset of a project how advanced architectural concepts may figure into a design, because *advanced* does not necessarily mean expensive or complex. Many technologies that were originally developed for high-end supercomputers and mainframes eventually found their way into consumer electronics and other less-expensive digital systems. This is why a digital engineer benefits from a broad understanding of advanced microprocessor and computing concepts—a wider palette of potential solutions enables a more creative and effective design process.

This chapter introduces a wide range of technologies that are alluded to in many technical specifications but are often not understood sufficiently to take full advantage of their potential. What is a 200-MHz superscalar RISC processor with a four-way set associative cache? Some people hear the term RISC and conjure up thoughts of high-performance computing. Such imagery is not incorrect, but RISC technology can also be purchased for less than one dollar. Caching is another big computer term that is more common than many people think.

An important theme to keep in mind is that microprocessors and the systems that they plug into are inextricably interrelated, and more so than simply by virtue of their common physical surroundings. The architecture of one directly influences the capabilities of the other. For this reason, the two need to be considered simultaneously during the design process. Among many other factors, this makes computer design an iterative process. One may begin with an assumption of the type of microprocessor required and then use this information to influence the broader system architecture. When system-level constraints and capabilities begin to come into focus, they feed back to the microprocessor requirements, possibly altering them somewhat. This cycle can continue for several iterations until a design is realized in which the microprocessor and its supporting peripherals are well matched for the application.

7.1 RISC AND CISC

One of the key features used to categorize a microprocessor is whether it supports cute simple instructions at a high rate—perhaps one instruction per cycle. Others believe that a microprocessor should execute more complex instructions at a lower rate.

Operand types add complexity to an instruction set when a single general operation such as addition can be invoked with many different addressing modes. Motorola's CISC 68000 contains a basic addition instruction, among other addition operations, that can be decoded in many different ways according to the specified addressing mode. Table 7.1 shows the format of the basic `ADD`/`ADDA`/`ADDX` instruction word. `ADD` is used for operations primarily on data registers. `ADDA` is used for operations primarily on address registers. `ADDX` is used for special addition operations that incorporate the ALU extended carry bit, X, into the sum. The instruction word references Register1 directly and an effective address (EA) that can represent another register or various types of indirect and indexed addressing modes.

TABLE 7.1 68000 ADD/ADDA/ADDX Instruction Word

<table>
<tr><td>Bit Position</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">Field</td><td colspan="4" rowspan="2">Opcode = 1101</td><td colspan="3" rowspan="2">Register1</td><td colspan="3" rowspan="2">Opmode</td><td colspan="6">Effective Address</td></tr>
<tr><td colspan="3">Mode</td><td colspan="3">Register2</td></tr>
</table>

As listed in Table 7.2, the opmode field defines whether the operands are 8-, 16-, or 32-bit quantities and identifies the source and destination operands. In doing so, it also implies certain subclasses of instructions: `ADD`, `ADDA`, or `ADDX`.

TABLE 7.2 68000 ADD/ADDA/ADDX Instruction Opmode Field

Opmode Value	Operand Width	Definition of Register1	Operation	Instruction Mapping
000	8	Dn	EA + Dn ⇒ Dn	ADD
001	16	Dn	EA + Dn ⇒ Dn	ADD
010	32	Dn	EA + Dn ⇒ Dn	ADD
100	8	Dn	Dn + EA ⇒ EA	ADD/ADDX
101	16	Dn	Dn + EA ⇒ EA	ADD/ADDX
110	32	Dn	Dn + EA ⇒ EA	ADD/ADDX
011	16	An	EA + An ⇒ An	ADDA
111	32	An	EA + An ⇒ An	ADDA

The main complexity is introduced by the EA fields as defined in Table 7.3. For those modes that map to multiple functions, additional identifying fields and operands are identified by one or more extension words that follow the instruction word. One of the more complex modes involves using an address register as a base address, adding a displacement to that base to calculate a fetch address, fetching the data at that address, adding another register to the retrieved value, adding another displacement, and then using the resulting address to fetch a final operand value. `ADD/ADDA/ADDX` is

In contrast to the 68000's CISC architecture, the MIPS family of microprocessors is one of the commercial pioneers of RISC. MIPS began as a 32-bit architecture with 32-bit instruction words and 32 general-purpose registers. In the 1990s the architecture was extended to 64 bits. MIPS instruction words are classified into three basic types: immediate (I-type), jump (J-type), and register (R-type). The original MIPS architecture supports four 32-bit addition instructions without any addressing mode permutations: add signed (ADD), add unsigned (ADDU), add signed immediate (ADDI), and add unsigned immediate (ADDIU). These instructions are represented by two types of instruction words, I-type and R-type, as shown in Table 7.4.

The immediate operations specify two registers and a 16-bit immediate operand: $R_T = R_S +$ Immediate. The other instructions operate on registers only and allow the programmer to specify three registers: $R_D = R_S + R_T$. If you want to add data that is in memory, that data must first be loaded into a register. Whereas a single 68000 instruction can fetch a word from memory, increment the associated pointer register, add the word to another register, and then store the result back into memory, a MIPS microprocessor would require separate instructions for each of these steps. This is in keeping with RISC concepts: use more simpler instructions to get the job done.

Instruction decode logic for a typical RISC microprocessor can be much simpler than for a CISC counterpart, because there are fewer instructions to decode and fewer operand complexities to recognize and coordinate. Generally speaking, a RISC microprocessor accesses data memory only with dedicated load/store instructions. Data manipulation instructions operate solely on internal registers and immediate operands. Under these circumstances, microprocessor engineers are able to heavily optimize their design in favor of the reduced instruction set that is supported. It turns out that not all instructions in a CISC microprocessor are used with the same frequency. Rather, there is a core set of instructions that are called most of the time, and the rest are used infrequently. Those that are used

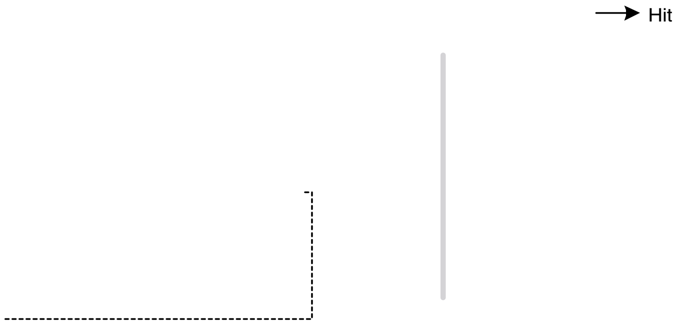
line is *clean* or *dirty.* When the line is eventually flushed, dirty lines must be written back to main memory in their entirety. Clean lines can be flushed without further action. While a write-back cache cannot absolutely eliminate the longer write latency of main memory, it can reduce the overall system impact of writes, because the microprocessor can perform any number of writes to the same cache line, and only a fixed write-back penalty results upon a flush.

The central problem in designing a cache is how to effectively hold many scattered blocks from a large main memory in a small cache memory. In a standard desktop PC, main memory may consist of 256 MB of DRAM, whereas the microprocessor's cache is 256 kB—a difference of three orders of magnitude! The concept of cache lines provides a starting point with a defined granularity to minimize the problem somewhat. Deciding on a 16-byte line size, for example, indicates that a 32-bit address space needs to be handled only as 2^{28} units rather than 2^{32} units. Of course, 2^{28} is still a very large number! Each cache line must have an associated tag and/or index that identifies the higher-order address bits that its contents represent (28 bits in this example). Different cache architectures handle these tags and indices to balance cache performance with implementation expense. The three standard cache architectures are *fully associative*, *direct mapped*, and *n-way set associative*.

A fully associative cache, shown in Fig. 7.2, breaks the address bus into two sections: the lower bits index into a selected cache line to select a byte within the line, and the upper bits form a tag that is associated with each cache line. Each cache line contains a valid bit to indicate whether it contains real data. Upon reset, the valid bits for each line are cleared to 0. When a cache line is loaded with data, its tag is set to the high-order address bits that are driven by the microprocessor. On subsequent transactions, those address bits are compared in parallel against every tag in the cache. A hit occurs when one tag matches the requested address, resulting in that line's data advancing to a final multiplexer where the addressed bytes are selected by the low-order address bits. A fully associative ing logic, and each match signal must be logically combined in a single location to generate a final hit/miss status flag.

A direct mapped cache, shown in Fig. 7.3, breaks the address bus into three sections; the lower bits retain their index function within a selected line, the middle bits select a single line from an array of uniquely addressable lines, and the upper bits form a tag to match the selected cache line. As before, each cache line contains a valid bit. The difference here is that each block of memory can only be mapped into one cache line—the one indexed by that block's middle address bits, A[15:4] in this example (indicating a 64-kB total cache size). During a cache miss, the controller determines which line is selected by the middle address bits, loads the line, sets the valid bit, and loads the line tag with the upper address bits. On subsequent accesses, the middle address bits select a single line whose tag is compared against the upper address bits. If they match, there is a cache hit. A direct mapped cache is much easier to implement as compared to a fully associative cache, because parallel tag matching is not required. Instead, the cache can be constructed with conventional memory and logic components using off-the-shelf RAM for both the tag and line data. The control logic can index into the RAM, check the selected tag for a match, and then take appropriate action. The disadvantage to a direct mapped cache is that, because of the fixed mapping of memory blocks to cache lines, certain data access patterns can cause rapid *thrashing*. Thrashing results when the microprocessor rapidly accesses alternate memory blocks. If the alternate blocks happen to map to the same cache line, the cache will almost always miss, because each access will result in a flush of the alternate memory block.

Given the simplicity of a direct mapped cache, it would be nice to strike a compromise between an expensive fully associative cache and a thrashing-sensitive direct mapped cache. The *n-way* set associative cache is such a compromise. As shown in Fig. 7.4, a two-way set associative cache is basically two direct mapped cache elements connected in parallel to reduce the probability of thrashing. More than two sets can be implemented to further reduce thrashing potential. Four-way and two-way set associative caches are very common in modern computers. Beyond four elements, the payback of thrashing avoidance to implementation complexity declines. The term *set* refers to the number of entries in each direct mapped element, 4,096 in this example. Here, the



cache has expanded to 128 kB in size using two 64 kB elements. If cost constraints dictate keeping a 64 kB cache, it would be preferable to reduce the set size to 2,048 rather than halve the line size, which is already at a practical minimum of 16 bytes. Reducing the set size to 2^{11} would increase the line tag to 17 bits to maintain a 32-bit address space representation. In a cache of this type, the controller can choose which of two (or four, or n) cache line locations to flush when a miss is encountered.

Deciding which line to flush when a cache miss occurs can be done in a variety of ways, and different cache architectures dictate varying approaches to this problem. A fully associative cache can place any main memory block into any line, while a direct mapped cache has only one choice for any given memory block. Three basic flush, or replacement, algorithms are as follows:

- *First-in-first-out (FIFO).* Track cache line ages and replace the oldest line.
- *Least-recently-used (LRU).* Track cache line usage and replace the line that has not been accessed longest.
- *Random.* Replace a random line.

A fully associative cache has the most flexibility in selecting cache lines and therefore the most complexity in tracking line usage. To perform either a FIFO or LRU replacement algorithm on a fully associative cache, each line would need a tracking field that could be updated and checked in parallel with all other lines. N-way set associative caches are the most interesting problems from a practical perspective, because they are used most frequently. Replacement algorithms for these caches are simplified, because the number of replacement choices is restricted to N. A two-way set associative cache can implement either FIFO or LRU algorithms with a single bit per line entry. For a FIFO algorithm, the entry being loaded anew has its FIFO bit cleared, and the other entry has its FIFO bit set, indicating that the other entry was loaded first. For an LRU algorithm, the entry being accessed at any given time has its LRU bit cleared, and the other has its LRU bit set, indicating that the other entry was used least recently. These algorithms and associated hardware are only slightly more complex for a four-way set associative cache that would require two status bits per line entry.

7.3 CACHES IN PRACTICE

Basic cache structures can be applied and augmented in different ways to improve their efficacy. One common manner in which caches are implemented is in pairs: an *I-cache* to hold instructions and a *D-cache* to hold data. It is not uncommon to see high-performance RISC microprocessors with integrated I/D caches on chip. Depending on the intended application, these integrated caches can be relatively small, each perhaps 8 kB to 32 kB in size. More often than not, these are two-way or four-way set associative caches. There are two key benefits to integrating two separate caches. First, instruction and data access patterns can combine negatively to cause thrashing on a single normal cache. If a software routine operates on a set of data whose addresses happen to overlap with the I-cache's index bits, alternate instruction and data fetch operations could cause repeated thrashing on the same cache lines. Second, separate caches can effectively provide a Harvard memory architecture from the microprocessor's local perspective. While it is often not practical to provide dual instruction and data memory interfaces at the chip level, as a result of excessive pin count, such considerations are much less restrictive within a silicon die. Separate I/D caches can feed from a shared chip-level memory interface but provide independent interfaces to the microprocessor core itself. This dual-bus arrangement increases the microprocessor's load/store bandwidth by enabling it to simultaneously fetch instructions and operands without conflict.

Dual I/D caches cannot guarantee complete independence of instruction and data memory, because, ultimately, they are operating through a shared interface to a common pool of main memory. The performance boost that they provide will be dictated largely by the access patterns of the applications running on the microprocessor. Such application-dependent performance is fundamental to all types of caches, because caches rely on locality to provide their benefits. Programs that scatter instructions and data throughout a memory space and alternately access these disparate locations will show less performance improvement with the cache. However, most programs exhibit fairly benefiregisters are unlike main memory, because memory just holds data and cannot modify it or take actions based on it.

Whereas caching an I/O region can cause system disruption, caching certain legitimate main memory regions can cause performance degradation due to thrashing. It may not be worth caching small routines that are infrequently executed, because the performance benefit of caching a quick maintenance routine may be small, and its effect on flushing more valuable cache entries may significantly slow down the application that resumes execution when the maintenance routine completes. A performance-critical application is often composed of a processing kernel along with miscellaneous initialization and maintenance routines. Most of the microprocessor time is spent executing kernel instructions, but sometimes the kernel must branch to maintenance routines for purposes such as loading or storing data. Unlike I/O regions that are inherently known to be cache averse, memory regions that should not be cached can only be known by the programmer and explicitly kept out of the cache.

Methods of excluding certain memory locations from the cache differ across system implementations. A cache controller will often contain a set of registers that enable the lockout of specific memory regions. On those integrated microprocessors that contain some address decoding logic as well as a cache controller, individual memory areas are configured into the decoding logic with programmable registers, and each is marked as cacheable or noncacheable. When the microprocessor performs a memory transaction, the address decoder sends a flag to the cache controller that tells it whether to participate in the transaction.

On the flip side of locking certain memory regions out of the cache, some applications can benefit from explicitly locking certain memory regions into the cache. Locking cache entries prevents the cache controller from flushing those entries when a miss occurs. A programmer may be able to lock a portion of the processing kernel into the cache to prevent arbitrary maintenance routines from disturbing the most frequently accessed sets of instructions and data.

Cache controllers perform burst transactions to main memory because of their multiword line architecture. Whether the cache is reading a new memory block on a cache miss or writing a dirty block back to main memory, its throughput is greatly increased by performing burst transfers rather than reading or writing a single word at a time. Normal memory transfers are executed by presenting an address and reading or writing a single unit of data. Each type of memory technology has its own associated latency between the address and data phases of a transaction. SRAM is characterized by very low access latency, whereas DRAM has a higher latency. Because main memory in most systems is composed of DRAM, single-unit memory transfers are inefficient, because each byte or word is penalized by the address phase overhead. Burst transfers, however, return multiple sequential data units while requiring only an initial address presentation, because the address specifies a starting address for a set of memory locations. Therefore, the overhead of the address phase is amortized across many data units, greatly increasing the efficiency of the memory system. Modern DRAM devices support burst transfers that nicely complement the cache controllers that often coexist in the same computer system.

As a result of cache subsystems being integrated onto the same chip along with high-performance microprocessors, the external memory interface is less a microprocessor bus and more a burst-mode cache bus. The microprocessor needs to be able to bypass the cache controller while accessing noncacheable memory locations or during boot-up when peripherals such as the cache controller have not yet been initialized. However, the external bus is often optimized for burst transfers, and absolute efficiency or simplicity when dealing with noncacheable locations may be a secondary concern to the manufacturer. If overall complexity can be reduced by giving up some performance in nonburst transfers, it may be worth the trade-off, because high performance microprocessors spend relatively little of their time accessing external memory directly. If they do, then something is wrong with the system design, because the microprocessor's throughput is sure to suffer.

Many microprocessors are designed to support multiple levels of caching to further improve performance. In this context, the cache that is closest to the microprocessor core is termed a *level-one*

often operate on larger sets of data or that must run many applications simultaneously may merit larger caches to offset less optimal locality properties. Computers meant to function as computation engines and network file servers can include several megabytes of L2 cache. Smaller embedded systems may suffice with only several kilobytes of L1 cache.

7.4 VIRTUAL MEMORY AND THE MMU

Multitasking operating systems execute multiple programs at the same time by assigning each program a certain percentage of the microprocessor's time and then periodically changing which instruction sequence is being executed. This is accomplished by a periodic timer interrupt that causes the OS kernel to save the state of the microprocessor's registers and then reload the registers with preserved state from a different program. Each program runs for a while and is paused, after which execution resumes without the program having any knowledge of having been paused. In this respect, the individual programs in a multitasking environment appear to have complete control over the computer, despite sharing the resources with others. Such a perspective makes programming for a multitasking OS easier, because the programmer does not have to worry about the infinite permutations of other applications that may be running at any given time. A program can be written as if it is the only application running, and the OS kernel sorts out the run-time responsibilities of making sure that each application gets fair time to run on the microprocessor.

Aside from fair access to microprocessor time, conflicts can arise between applications that accidentally modify portions of each other's memory—either program or data. How does an application know where to locate its data so that it will not disturb that of other applications and so that it will not be overwritten? There is also the concern about system-wide fault tolerance. Even if not malicious, programs may have bugs that cause them to crash and write data to random memory locations. In such an instance, one errant application could bring down others or even crash the OS if it overwrites program and data regions that belong to the OS kernel. The first problem can be addressed with the honor system by requiring each application to dynamically request memory allocations at run time from the kernel. The kernel can then make sure that each application is granted an exclusive region of memory. However, the second problem of errant writes requires a hardware solution that can physically prevent an application from accessing portions of memory that do not belong to it.

Virtual memory is a hardware enforced and software configured mechanism that provides each application with its own private memory space that it can use arbitrarily. This virtual memory space can be as large as the microprocessor's addressing capability—a full 4 GB in the case of a 32-bit microprocessor. Because each application has its own exclusive virtual memory space, it can use any portion of that space that is not otherwise restricted by the kernel. Virtual memory frees the programmer from having to worry about where other applications may locate their instructions or data, because applications cannot access the virtual memory spaces of others. In fact, operating systems that support virtual memory may simplify the physical structure of programs by specifying a fixed starting address for instructions, the local stack, and data. UNIX is an example of an OS that does this. Each application has its instructions, stack, and data at the same virtual addresses, because they have separate virtual memory spaces that are mutually exclusive and, therefore, not subject to conflict.

Clearly, multiple programs cannot place different data at the same address or each simultaneously occupy the microprocessor's entire address space. The OS kernel configures a hardware *memory management unit* dled on a process basis rather than an application basis, because it is possible for an application to consist of multiple semi-independent processes. The high-order address bits referenced by each instruction form the *virtual page number* (VPN). The PID and VPN are combined to uniquely map to a physical address set aside by the kernel as shown in Fig. 7.5. Low-order address bits represent offsets that directly index into mapped pages in physical memory. The mapping of virtual memory pages into physical memory is assigned arbitrarily by the OS kernel. The kernel runs in real memory rather than in virtual memory so that it can have direct access to the computer's physical resources to allocate memory as individual processes are executed and then terminated.

Despite each process having a 4-GB address space, virtual memory can work on computers with just megabytes of memory, because the huge virtual address spaces are sparsely populated. Most processes use only a few hundred kilobytes to a few megabytes of memory and, therefore, multiple processes that collectively have the potential to reference tens of gigabytes can be mapped into a much smaller quantity of real memory. If too many processes are running simultaneously, or if these processes start to consume too much memory, a computer can exhaust its physical memory resources, thereby requiring some intervention from the kernel to either suspend a process or handle the problem in some other way.

When a process is initiated, or *spawned*, it is assigned a PID and given its own virtual memory space. Some initial pages are allocated to hold its instructions and whatever data memory the process needs available when it begins. During execution, processes may request more memory from the kernel by calling predefined kernel memory management routines. The kernel will respond by allocating a page in physical memory and then returning a pointer to that page's virtual mapping. Likewise, a process can free a memory region when it no longer needs it. Under this circumstance, the kernel will remove the mapping for the particular pages, enabling them to be reallocated to another process, or the same process, at a later time. Therefore, the state of memory in a typical multitasking OS is quite dynamic, and the routines to manage memory must be implemented in software because of their complexity and variability according to the platform and the nature of processes running at any given time.

Not all mapped virtual memory pages have to be held in physical RAM at the same time. Instead, the total virtual memory allocation on a computer can spill over into a secondary storage medium such as a hard drive. The hard drive will be much slower than DRAM, but not every memory page in every process is used at the same time. When a process is first loaded, its entire instruction image is typically loaded into virtual memory. However, it will take some time for all of those instructions to memory can be stored on the hard drive without incurring a performance penalty. When those instructions are ready to be executed, the OS kernel will have to transfer the data into physical memory. This slows the system down but makes it more flexible without requiring huge quantities of DRAM. Part of the kernel's memory management function is to decide which virtual pages should be held in DRAM and which should be *swapped* out to the disk. Pages that have not been used for a while can be swapped out to make room for new pages that are currently needed. If a process subsequently accesses a page that has been moved to the disk, that page can be swapped back into DRAM to replace another page that is not needed at the time. A computer with 256 MB of DRAM could, for example, have a 512-MB swap file on its hard drive, enabling processes to share a combined 768 MB of used virtual memory.

This scheme of expanding virtual memory onto a disk effectively turns the computer's DRAM into a large cache for an even larger disk-based memory. As with all caches, certain behavioral characteristics exist. A virtual memory page that is not present in DRAM is effectively a cache miss with a large penalty, because hard disks are much slower than DRAM. Such misses are called *page faults*. The MMU detects that the requested virtual memory address from a particular PID is not present in DRAM and causes an exception that must be handled by the OS kernel. Instead of performing a cache line fill and flush, it is the kernel's responsibility to swap pages to and from the disk. For a virtual memory system to function with reasonable performance, the *working set* of memory across all the processes running should be able to fit into the computer's physical memory. The working set includes any instructions and data that are accessed within a local time interval. This is directly analogous to a microprocessor cache's exploitation of locality. Processes with good locality characteristics will do well in a cache and in a virtual memory system. Processes with poor locality may result in thrashing as many sequential page faults are caused by random accesses throughout a large virtual memory space.

The virtual to physical address mapping process is guided by the kernel using a *page table*, which can take various forms but must somehow map each PID/VPN combination to either a physical memory page or one located on the disk drive's swap area. Virtual page mapping is illustrated in Fig. 7.6, assuming 4-kB pages, a 32-bit address space, and an 8-bit PID. In addition to basic mapping information, the page table also contains status information, including a dirty bit that indicates when a page held in memory has been modified. If modified, the page must be saved to the disk before being flushed to make room for a new virtual page. Otherwise, the page can be flushed without further action.

Given a 4-kB page size and a 32-bit address space, each process has access to 2^{20} = 1,048,576 pages. With 256 PIDs, a brute-force page table would contain more than 268 million entries! There are a variety of schemes to reduce page table size, but there is no escaping the fact that a page table

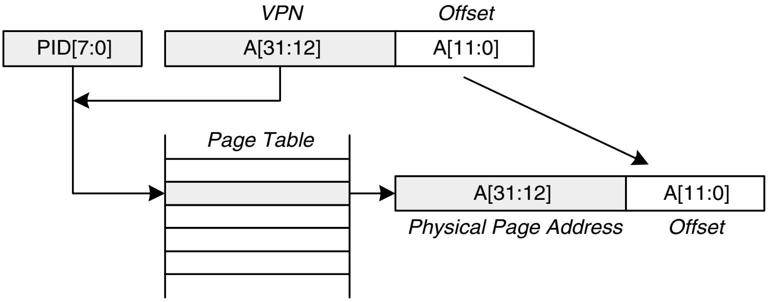


FIGURE 7.6 Virtual page mapping.

delays. Throughput can be enhanced in a serial manner by trying to execute a desired function faster. If each function is executed at a faster clock frequency, more functions can be executed in a given time period. An alternative parallel approach can be taken whereby multiple functions are executed simultaneously, thereby improving performance over time. These two approaches can be complementary in practice. Different logic implementations make use of serial and parallel enhancement techniques in the proportions and manners that are best suited to the application at hand.

A logic function is represented by a set of Boolean equations that are then implemented as discrete gates. During one clock cycle, the inputs to the equations are presented to a collection of gates via a set of input flops, and the results are clocked into output flops on the next rising edge. The propagation delays of the gates and their interconnecting wires largely determine the shortest clock period at which the logic function can reliably operate.

Pipelining, called *superpipelining* when taken to an extreme, is a classic serial throughput enhancement technique. Pipelining is the process of breaking a Boolean equation into several smaller equations and then calculating the partial results during sequential clock cycles. Smaller equations require fewer gates, which have a shorter total propagation delay relative to the complete equation. The shorter propagation delay enables the logic to run faster. Instead of calculating the complete result in a single 40 ns cycle, for example, the result may be calculated in four successive cycles of 10 ns each. At first glance, it may not seem that anything has been gained, because the calculation still takes 40 ns to complete. The power of pipelining is that different stages in the pipeline are operating on different calculations each cycle. Using an example of an adder that is pipelined across four cycles, partial sums are calculated at each stage and then passed to the next stage. Once a partial sum is passed to the next stage, the current stage is free to calculate the partial sum of a completely new addition operation. Therefore, a four-stage pipelined adder takes four cycles to produce a result, but it can work on four separate calculations simultaneously, yielding an average throughput of one calculation every cycle—a four-times throughput improvement.

Pipelining does not come for free, because additional logic must be created to handle the complexity of tracking partial results and merging them into successively more complete results. Pipelining a 32-bit unsigned integer adder can be done as shown in Fig. 7.8 by adding eight bits at a time and then passing the eight-bit sum and carry bit up to the next stage. From a Boolean equation perspective, each stage only incurs the complexity of an 8-bit adder instead of a 32-bit adder, enabling it to run faster. An array of pipeline registers is necessary to hold the partial sums that have been calculated by previous stages and the as-yet-to-be-calculated portions of the operands. The addition results ripple through the pipeline on each rising clock edge and are accumulated into a final 32-bit result as operand bytes are consumed by the adders. There is no feedback in this pipelined adder, meaning that, once a set of operands passes through a stage, that stage no longer has any involvement in the operation and can be reused to begin or continue a new operation.

Pipelining increases the overall throughput of a logic block but does not usually decrease the calculation latency. High-performance microprocessors often take advantage of pipelining to varying degrees. Some microprocessors implement superpipelining whereby a simple RISC instruction may have a latency of a dozen or more clock cycles. This high degree of pipelining allows the microprocessor to execute an average of one instruction each clock cycle, which becomes very powerful at operating frequencies measured in hundreds of megahertz and beyond.

Superpipelining a microprocessor introduces complexities that arise from the interactions between consecutive instructions. One instruction may contain an operand that is calculated by the previous instruction. If not handled correctly, this common circumstance can result in the wrong value being used in a subsequent instruction or a loss of performance where the pipeline is frequently stalled to allow one instruction to complete before continuing with others. Branches can also cause havoc with a superpipelined architecture, because the decision to take a conditional branch may nullify the few instructions that have already been loaded into the pipeline and partially executed. De-

Managing parallel execution units in a superscalar microprocessor is a complex task, because the microprocessor wants to execute instructions as fast as they can be fetched—yet it must do so in a manner consistent with the instructions' serial interdependencies. These dependencies can become more complicated to resolve when superscalar and superpipelining techniques are combined to create a microprocessor with multiple execution units, each of which is implemented with a deep pipeline. In such chips, the instruction decode logic handles the complex task of examining the pipelines of the execution units to determine when the next instruction is free of dependencies, allowing it to begin execution.

Related to superpipelining and superscalar methods are the techniques of branch prediction, speculative execution, and instruction reordering. Deep pipelines are subject to performance-degrading flushes each time a branch instruction comes along. To reduce the frequency of pipeline flushes due to branch instructions, some microprocessors incorporate branch prediction logic that attempts to make a preliminary guess as to whether the branch will be taken. These guesses are made based on the history of previous branches. The exact algorithms that perform branch prediction vary by implementation and are not always disclosed by the manufacturer, to protect their trade secrets. When the branch prediction logic makes its guess, the instruction fetch and decode logic can speculatively execute the instruction stream that corresponds to the predicted branch result. If the prediction logic is correct, a costly pipeline flush is avoided. If the prediction is wrong, performance will temporarily degrade until the pipeline can be restarted. Hopefully, a given branch prediction algorithm improves performance rather than degrading it by having a worse record than would exist with no prediction at all!

The problem with branch prediction is that it is sometimes wrong, and the microprocessor must back out of any state changes that have resulted from an incorrectly predicted branch. Speculative execution can be taken a step farther in an attempt to eliminate the penalty of a wrong branch prediction by executing both possible branch results. To do this, a superscalar architecture is needed that has enough execution units to speculatively execute extra instructions whose results may not be used. It is a foregone conclusion that one of the branch results will not be valid. There is substantial complexity involved in such an approach because of the duplicate hardware that must be managed and the need to rapidly swap to the correct instruction stream that is already in progress when the result of a branch is finally known.

A superscalar microprocessor will not always be able to keep each of its execution units busy, because of dependencies across sequential instructions. In such a case, the next instruction to be pushed into the execution pipeline must be held until an in-progress instruction completes. Instruction reordering logic reduces the penalty of such instruction stalls by attempting to execute instructions outside the order in which they appear in the program. The microprocessor can prefetch a set of instructions ahead of those currently executing, enabling it to look ahead in the sequence and determine whether a later instruction can be safely executed without changing the behavior of the instruction stream. For such reordering to occur, an instruction must not have any dependencies on those that are being temporarily skipped over. Such dependencies include not only operands but branch possibilities as well. Reordering can occur in a situation in which the ALUs are busy calculating results that are to be used by the next instruction in the sequence, and their latencies are preventing the next instruction from being issued. A load operation that is immediately behind the stalled instruction can be executed out of order if it does not operate on any registers that are being used by the instructions ahead of it. Such reordering boosts throughput by taking advantage of otherwise idle execution cycles.

All of the aforementioned throughput improvement techniques come at a cost of increased design complexity and cost. However, it has been widely noted that the cost of a transistor on an IC is asymptotically approaching zero as tens of millions of transistors are squeezed onto chips that cost only several hundred dollars. Once designed, the cost of implementing deep pipelines, multiple execution units, and the complex logic that coordinates the actions of both continues to decrease over time.

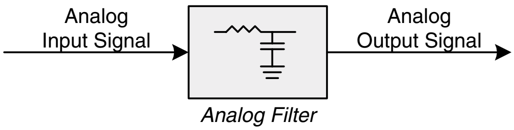
As can be readily observed from Table 7.5, very large and very small numbers can be represented because of the wide ranges of exponents provided in the various formats. However, the representation of 0 seems rather elusive with the requirement that the mantissa always have a leading 1. Values including 0 and infinity are represented by using the two-exponent values that are not supported for normal numbers: 0 and $2^n - 1$. In the case of the single-precision format, these values are 0x00 and 0xFF.

An exponent field of 0x00 is used to represent numbers of very small magnitude, where the interpreted exponent value is fixed at the minimum for that format: –126 for single precision. With a 0 exponent field, the mantissa's definition changes to a number greater than or equal to 0 and less than 1. Smaller numbers can now be represented, though with decreasing significant figures, because magnitude is now partially represented by the significand field. For example, 101×2^{-130} is expressed as 0.0101×2^{-126}. Such special-case numbers are *denormalized,* because their mantissas defy the *normalized* form of being greater than or equal to 1 and less than 2. Zero can now be expressed by setting the significand to 0 with the result that $0 \times 2^{-126} = 0$. The presence of the sign bit produces two representations of zero, positive and negative, that are mathematically identical.

Setting the exponent field to 0xFF (in single precision) is used to represent either infinity or an undefined value. Positive and negative infinity are represented by setting the significand field to 0 and using the appropriate sign bit. When the exponent field is 0xFF and the significand field is nonzero, the representation is "not a number," or *NaN*. Examples of computations that may return NaN are $0 \div 0$ and $\infty \div \infty$.

7.7 DIGITAL SIGNAL PROCESSORS

Microprocessor architectures can be optimized for increased efficiency in certain applications through the inclusion of special instructions and execution units. One major class of application-specific microprocessors is the *digital signal processor*, or DSP. DSP entails a microprocessor mathematically manipulating a sampled analog signal in a way that emulates transformation of that signal by discrete analog components such as filters or amplifiers. To operate on an analog signal digitally, the analog signal must be sampled by an analog-to-digital converter, manipulated, and then reconstructed with a digital-to-analog converter. A rough equivalency of digital signal processing versus conventional analog transformation is shown in Fig. 7.11 in the context of a simple filter.



more with a 32- or 64-bit data path, such a seemingly powerful memory array may just barely be able to keep up.

While bandwidth can be increased by widening the interface, random access latency does not go away. Therefore, there is more to a memory array than its raw size. The bandwidth of the array, which is the product of its interface frequency and width, and its latency are important metrics in understanding the impact of cache misses, especially when dealing with applications that exhibit poor locality.

Caching reduces the negative effect of high random access latencies on a microprocessor's throughput. However, caches and wide arrays cannot completely balance the inequality between the bandwidth and latency that the microprocessor demands and that which is provided by SDRAM technology. Cache size, type, and latency and main memory bandwidth are therefore important metrics that contribute to overall system performance. An application's memory characteristics determine how costly a memory architecture is necessary to maintain adequate performance. Applications that operate on smaller sets of data with higher degrees of locality will be less reliant on a large cache and fast memory array, because they will have fewer cache misses. Those applications with opposite memory characteristics will increase the memory architecture's effect on the computer's overall performance. In fact, by the nature of the application being run, caching effects can become more significant than the microprocessor's core clock frequency. In some situations, a 500-MHz microprocessor with a 2-MB cache can outperform a 1-GHz microprocessor with a 256-kB cache. It is important to understand these considerations because money may be better spent on either a faster microprocessor or a larger cache according to the needs of the intended applications.

I/O performance affects system throughput in two ways: the latency of executing transactions and the degree to which such execution blocks the microprocessor from performing other work. In a computer in which the microprocessor operates with a substantially higher bandwidth than individual I/O interfaces, it is desirable to decouple the microprocessor from the slower interface as much as possible. Most I/O controllers provide a natural degree of decoupling. A typical UART, for example, absorbs one or more bytes in rapid succession from a microprocessor and then transmits them at a slower serial rate. Likewise, the UART assembles one or more whole incoming bytes that the microprocessor can read at an instantaneous bandwidth much higher than the serial rate. Network and disk adapters often contain buffers of several kilobytes that can be rapidly filled or drained by the microprocessor. The microprocessor can then continue with program execution while the adapter logic handles the data at whatever lower bandwidth is inherent to the physical interface.

Inherent decoupling provided by an I/O controller is sufficient for many applications. When dealing with very I/O-intensive applications, such as a large server, multiple I/O controllers may interact with each other and memory simultaneously in a multimaster bus configuration. In such a context, the microprocessor sets up block data transfers by programming multiple I/O and DMA controllers and then resumes work processing other tasks. Each I/O and DMA controller is a potential bus master that can arbitrate for access to the memory system and the I/O bus (if there is a separate I/O bus). As the number of simultaneous bus masters increases, contention can develop, which may cause performance degradation resulting from excessive waiting time by each potential bus master. This contention can be reduced by modifying the I/O bus architecture. A first step is to decouple the I/O bus from the memory bus into one or more segments, enabling data transfers within a given I/O segment to proceed without conflicting with a memory transfer or one contained within other I/O segments. PCI is an example of such a solution. At a more advanced level, the I/O system can be turned into a switched network in which individual I/O controllers or small segments of I/O controllers are connected to a dedicated port on an I/O switch that enables each port to communicate with any other port simultaneously insofar as multiple ports do not conflict for access to the same port. This is a fairly expensive solution that is implemented in high-end servers for which I/O performance is a key contributor to overall system throughput.

CHAPTER 8
High-Performance Memory Technologies

Memory is an interesting and potentially challenging portion of a digital system design. One of the benefits of decades of commercial solid-state memory development is the great variety of memory products available for use. Chances are that there is an off-the-shelf memory product that fits your specific application. A downside to the modern, ever-changing memory market is rapid obsolescence of certain products. DRAM is tied closely to the personal computer market. The best DRAM values are those devices that coincide with the sweet spot in PC memory configurations. As the high-volume PC market moves on to higher-density memory ICs, that convenient DRAM that you used in your designs several years ago may be discontinued so that the manufacturer can retool the factory for parts that are in greater demand.

Rapid product development means that memory capabilities improve dramatically each year. Whether it's higher density or lower power that an application demands, steady advances in technology put more tools at an engineer's disposal. SRAM and flash EPROM devices have more stable production lives than DRAM. In part, this is because they are less dependent on the PC market, which requires ever increasing memory resources for ever more complex software applications.

Memory is a basic digital building block that is used for much more than storing programs and data for a microprocessor. Temporary holding buffers are used to store data as it is transferred from one interface to another. There are many situations in networking and communication systems where a block of data arrives and must be briefly stored in a buffer until the logic can figure out exactly what to do with it. Lookup tables are another common use for memory. A table may store precomputed terms of a complex calculation so that a result can be rapidly determined when necessary. This chapter discusses the predominant synchronous memory technologies, SDRAM and SSRAM, and closes with a presentation of CAM, a technology that is part RAM and part logic.

No book can serve as an up-to-date reference on memory technology for long, as a result of the industry's rapid pace. This chapter discusses technologies and concepts that are timeless, but specifics of densities, speeds, and interface protocols change rapidly. Once you have read and understood the basics of high-performance memory technologies, you are encouraged to browse through the latest manufacturers' data sheets to familiarize yourself with the current state of the art. Corporations such as Cypress, Hynix, Infineon, Micron, NEC, Samsung, and Toshiba provide detailed data sheets on their web sites that are extremely useful for self-education and selecting the right memory device to suit your needs.

8.1 SYNCHRONOUS DRAM

As system clock frequencies increased well beyond 50 MHz, conventional DRAM devices with asynchronous interfaces became more of a limiting factor in overall system performance. Asynchro-

Once the CAS latency has passed, data begins to flow on every clock cycle. Data will flow for as long as the specified burst length. In Fig. 8.2, the standard burst length is four words. This parameter is configurable and adds to the flexibility of an SDRAM. The controller is able to set certain parameters at start-up, including CAS latency and burst length. The burst length then becomes the default unit of data transfer across an SDRAM interface. Longer transactions are built from multiple back-to-back bursts, and shorter transactions are achieved by terminating a burst before it has completed. SDRAMs enable the controller to configure the standard burst length as one, two, four, or eight words, or the entire row. It is also possible to configure a long burst length for reads and only single-word writes. Configuration is performed with the mode register set (MRS) command by asserting the three primary control signals and driving the desired configuration word onto the address bus.

As previously mentioned, DQM signals function as an output disable on a read. The DQM bus (a single signal for SDRAMs with data widths of eight bits or less) follows the CAS* timing and, therefore, leads read data by the number of cycles defined in the CAS latency selection. The preceding read can be modified as shown in Fig. 8.3 to disable the two middle words.

In contrast, write data does not have an associated latency with respect to CAS*. Write data begins to flow on the same cycle that the WR/WRA command is asserted, as shown in Fig. 8.4. This

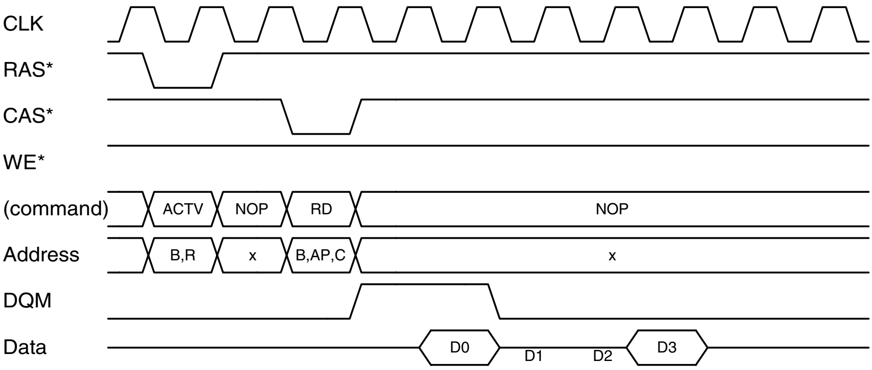


FIGURE 8.3 Four-word SDRAM burst read with DQM disable (CL = 2, BL = 4).

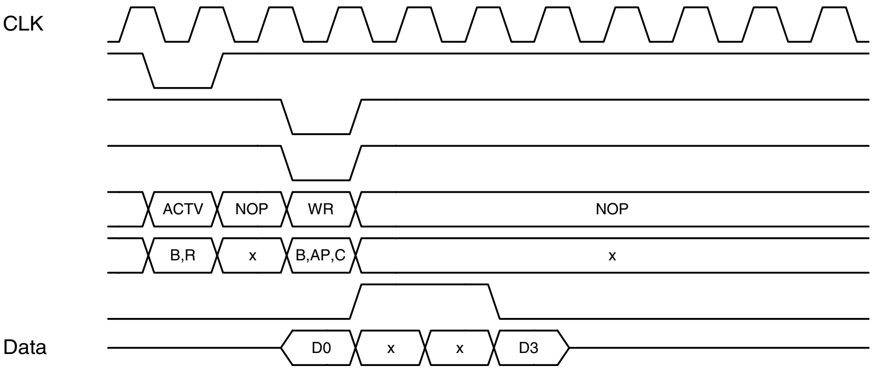


FIGURE 8.4 Four-word SDRAM burst write with DQM masking (BL = 4).

Periodic refresh is a universal requirement of DRAM technology, and SDRAMs are no exception. An SDRAM device may contain 4,096 rows per bank (or 8,192, depending on its overall size) with the requirement that all rows be refreshed every 64 ms. Therefore, the controller has the responsibility of ensuring that 4,096 (or 8,192) refresh operations are carried out every 64 ms. Refresh commands can be evenly spaced every 15.625 µs (or 7.8125 µs), or the controller might wait until a certain event has passed and then rapidly count out 4,096 (or 8,192) refresh commands. Different SDRAM devices have slightly differing refresh requirements, but the means of executing refresh operations is standardized. The first requirement is that all banks be precharged, because the auto-refresh (REF) command operates on all banks at once. An internal refresh counter keeps track of the next row across each bank to be refreshed when a REF command is executed by asserting RAS* and CAS* together.

It can be easy to forget the asynchronous timing requirements of the DRAM core when designing around an SDRAM's synchronous interface. After a little time spent studying state transition tables and command sets, the idea that an asynchronous element is lurking in the background can become an elusive memory. Always be sure to verify that discrete clock cycle delays conform to the nanosecond timing specifications that are included in the SDRAM data sheet. The tricky part of these timing specifications is that they affect a system differently, depending on the operating frequency. At 25 MHz, a 20-ns time delay is less than one cycle. However, at 100 MHz, that delay stretches to two cycles. Failure to recognize subtle timing differences can cause errors that may manifest themselves as intermittent data corruption problems, which can be very time consuming to track down.

SDRAM remains a mainstream memory technology for PCs and therefore is manufactured in substantial volumes by multiple manufacturers. The SDRAM market is a highly competitive one, with faster and denser products appearing regularly. SDRAMs are commonly available in densities ranging from 64 to 512 Mb in 4, 8, and 16-bit wide data buses. Older 16-Mb parts are becoming harder to find. For special applications, 32-bit wide devices are available, though sometimes at a slight premium as a result of lower overall volumes.

8.2 DOUBLE DATA RATE SDRAM

Conventional SDRAM devices transfer one word on the rising edge of each clock cycle. At any given time, there is an upper limit on the clock speed that is practical to implement for a board-level interface. When this level of performance proves insufficient, *double data rate* (DDR) SDRAM devices can nearly double the available bandwidth by transferring one word on both the rising and falling edges of each clock cycle. In doing so, the interface's clock speed remains constant, but the data bus effectively doubles in frequency. Functionally, DDR and single data rate (SDR) devices are very similar. They share many common control signals, a common command set, and a rising-edge-only control/address interface. They differ not only in the speed of the data bus but also with new DDR data control signals and internal clocking circuitry to enable reliable circuit design with very tight timing margins. Figure 8.6 shows the DDR SDRAM structure.

A DDR SDRAM contains an internal data path that is twice the width of the external data bus. This width difference allows the majority of the internal logic to run at a slower SDR frequency while delivering the desired external bandwidth with half as many data pins as would be required

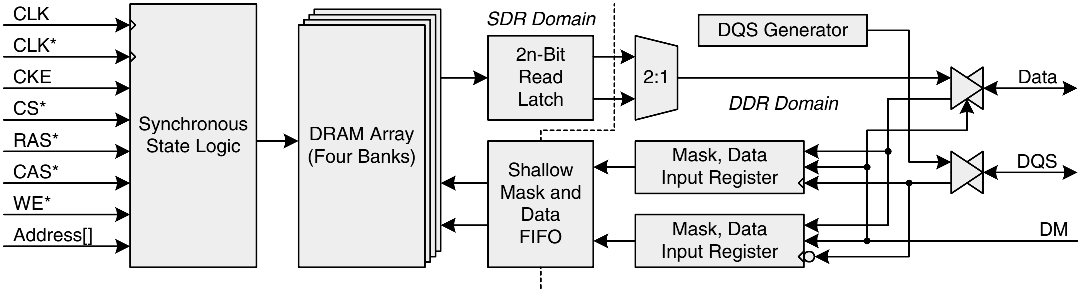


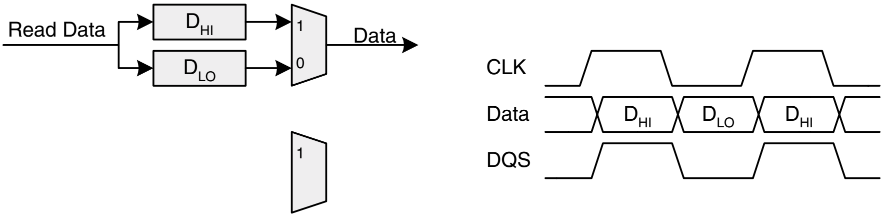
FIGURE 8.6 Basic DDR SDRAM architecture.

ble the speed of the control interface, because an SDRAM is almost always used in burst mode where the rate of commands is significantly less than the rate of data transferred.

The data interface contains a mask that has been renamed to DM and a new data strobe signal, DQS. DM functions as DQM does in an SDR device but operates at DDR to match the behavior of data. DQS is a bidirectional clock that is used to help time the data bus on both reads and writes. On writes, DQS, DM, and data are inputs and DQS serves as a clock that the SDRAM uses to sample DM and data. Setup and hold times are specified relative to both the rising and falling edges of DQS, so DQS transitions in the middle of the data valid window. DQS and data are outputs for reads and are collectively timed relative to CLK/CLK*. DQS transitions at roughly the same time as data and so it transitions at the beginning of the data valid window.

When reading, 2n bits are fetched from the DRAM array on the CLK domain and are fed into a 2:1 multiplexer that crosses the SDR/DDR clock domain. In combination with a DQS generator, the multiplexer is cycled at twice the CLK frequency to yield a double rate interface. This scheme is illustrated schematically in Fig. 8.7. Because DQS and data are specified relative to CLK/CLK* on reads, the memory controller can choose to clock its input circuitry with any of the strobe or clock signals according to the relevant timing specifications. Writes function in a reverse scheme by stacking two n-bit words together to form a 2n-bit word in the DRAM's CLK domain. Two registers are each clocked alternately on the rising and falling edges of DQS, and their contents are then transferred to a shallow write FIFO. A FIFO is necessary to cross from the DQS to CLK domains reliably as a result of skew between the two signals.

Tight timing specifications characterize DDR SDRAM because of its high-speed operation: a 333-MHz data rate with a 167-MHz clock is not an uncommon operating frequency. For reliable operation, careful planning must be done at the memory controller and in printed circuit board design to ensure that data is captured in as little as 1.5 ns (for a 333/167-MHz DDR SDRAM). These high-



speed data buses are treated as *source-synchronous* rather than synchronous. A source-synchronous bus is one where a local clock is generated along with data and routed on the circuit board with the data signals. The clock and data signals are length-matched to a certain tolerance to greatly reduce the skew between all members of the bus. In doing so, the timing relationships between clock and data are preserved almost exactly as they are generated by the sending device. A source-synchronous bus eliminates system-level skew problems that result from clocks and data signals emanating from different sources and taking different paths to their destinations. Treating the DDR SDRAM data bus source-synchronously as shown in Fig. 8.7 guarantees that the data valid window provided by the driver will be available to the load. Likewise, because DQS is bidirectional, the SDRAM will obtain the same timing benefit when accepting write-data from the memory controller.

Methods vary across DDR SDRAM implementations. While the SDRAM requires a fixed relationship between DQS and data for writes, the memory controller may use either DQS or a source-synchronous version of CLK with which to time read data. DQS must be used for the fastest applications, because it has a closer timing relationship relative to data. The usage of DQS adds some complexity, because it is essentially a bidirectional clock. There are also multiple DQS signals in most applications, because one DQS is present for every eight bits of data.

Some applications may be able to use CLK/CLK* to register read data. The memory controller typically drives CLK/CLK* to the SDRAM along with address and control signals in a source-synchronous fashion. To achieve a source-synchronous read data bus, a skewed version of CLK/CLK* is necessary that is in phase with the returned data so that the memory controller sees timing as shown in Fig. 8.7. This skew is the propagation delay through the wires that carry the clocks from the memory controller to the SDRAM. These skews are illustrated in Fig. 8.8a, and the associated wiring implementation is shown in Fig. 8.8b. CLK´ and CLK´* are the clocks that have been skewed by propagation delay through the wiring. A source-synchronous read-data bus is achieved by generating a second pair of clocks that are identical to the main pair and then by matching their lengths to the sum of the wire lengths to and from the SDRAM. The first length component cancels out the propagation delay to the SDRAM, and the second length component maintains timing alignment, or phase, with the data bus.

With the exception of a faster data bus, a DDR SDRAM functions very much like a conventional SDRAM. Commands are issued on the rising edge of CLK and are at a single data rate. Because of the internal 2n-bit architecture, a minimum burst size of two words is supported. The other burst length options are four or eight words. To read or write a single word, DM must be used to mask or

caches and other applications that operate using bursts. An SSRAM contains one or more control signals that defines whether a memory cycle uses an externally supplied address or an internally latched address and counter. When a burst transfer is desired, the memory controller asserts a control signal to load the internal burst counter and then directs the SSRAM to use that incrementing count value for the three subsequent cycles. Bursts are supported for both reads and writes. The two-bit burst counter can be configured in one of two increment modes: linear and interleaved. Linear increment is a simple binary counter that wraps from a terminal value of 11 back to 00. Bursts can be initiated at any address, so, if the burst begins at A[1:0] = 10, the counter will count 10, 11, 00, and 01 to complete the burst. Interleaved mode forces the data access pattern into two pairs where each pair contains an odd and even address with A[1] held constant as shown in Table 8.2. Interleaving can benefit implementations that access words in specific pairs.

TABLE 8.2 SSRAM Interleaved Burst Addressing

Initial Value of A[1:0] Supplied Externally	Second Address Generated Internally	Third Address Generated Internally	Fourth Address Generated Internally
00	01	10	11
01	00	11	10
10	11	00	01
11	10	01	00

[illegible]-through and pipelined SSRAMs fall into two more categories: normal and *zero-bus turn-[illegible]*®(ZBT)[*]. Normal SSRAMs exhibit differing read and write latencies: write data can be as-s[illegible] the same cycle as the address and write enable signals, but reads have one to two cycles of la[illegible] pending on the type of device being used. Under conditions of extended reads or writes, the [illegible]M can perform a transfer each clock cycle, because the latency of sequential commands (all [illegible] all writes) remains constant. When transitioning from writing to reading, however, the asym[illegible] auses idle time on the SSRAM data bus because of the startup latency of a read command. [illegible]d command is issued in the cycle immediately following the write, and read data becomes [illegible]e one or two cycles later. If an application performs few bus turnarounds because its tends to [illegible]tely execute strings of reads followed by writes, the loss of a few cycles here and there is p[illegible]y not a concern. However, some applications continually perform random read/write transactio[illegible]emory and may lose necessary bandwidth each time a bus turnaround is performed.

ZBT dev[illegible]olve the turnaround idle problem by enforcing symmetrical delays between address and data, reg[illegible]ss of whether the transaction is a read or write. This fixed relationship means that any comman[illegible] follow any other command without forced idle time on the data bus. Flow-through ZBT devices p[illegible]ent data on the first clock edge following the corresponding address/command. Pipelined ZBT [illegible]RAMs present data on the second clock edge following the corresponding address/command as sh[illegible]n in Fig. 8.12. As with normal SSRAMs, higher clock frequencies are possible with [illegible]vices, albeit at the expense of additional read latency.

[illegible] frequent read/write transitions. One ex-[illegible]M and control logic. A ge-

Such lookup tables are common in communication systems where decisions are made and statistics gathered according to the unique tags and network addresses present in each packet's header. When the size of a tag is bounded at a manageable width, a conventional memory array can be used to implement a lookup table. However, as tags grow to 16, 32, 64, 128, or more bits, the required memory size becomes quite impractical. The example in Fig. 8.17 would require 8 GB of memory if the tag width increased from 8 to 32 bits! If all 2^{32} tag permutations need to be accounted for independently, there would be no avoiding a large memory array. However, the majority of such lookup table applications handle a small fraction of the total set of permutations. The working set of tags sparsely populates the complete defined set of tags. So the question becomes how to rapidly index into a memory array with an N-bit tag where the array size is much less than 2^N.

A *content addressable memory* (CAM) solves this problem with an array of fully associative tags and optional corresponding data entries as shown in Fig. 8.18. Instead of decoding 2^N unique locations based on an N-bit tag, each CAM entry simultaneously matches its own tag to the one presented. The entry whose tag matches is the one that presents its associated data at the output and the one that can have its data modified as well. Alternatively, a CAM may simply return the index of the matched or winning entry in the array, if the specific device does not have any data associated with each entry. There is substantial overhead in providing each entry with a unique tag and matching associate a pair of tag bits with each actual tag bit. This two-bit structure allows the creation of a third "don't care" state, X. A ternary CAM is more flexible than a binary CAM, because it can match portions of a tag rather than all bits. In networking applications, this is very useful, because similar operations are often performed on groups of addresses (tags) from common destinations. It is as if the post office wanted to sort out all letters being sent to ZIP codes 11230 through 11239. A ternary CAM would be able to match the pattern 1123X with a single entry. In contrast, a binary CAM would require ten redundant entries to perform the same job.

A ternary CAM is often used to implement rather complex lookup tables with searches prioritized according to the number of X bits in each tag. Using the ZIP code example, it is possible that a post office would want to perform two overlapping searches. It may want to sort all ZIP codes from 11230 through 11239 into a particular bin, except for 11234, which should be sorted into its own bin. A ternary CAM could be setup with two overlapping entries: 11234 and 1123X. To ensure that the 11234 entry always matched ahead of the 1123X entry, it would be necessary to verify proper setup of the specific CAM being used. A ternary CAM may have a rule that the lowest or highest winning entry in the array wins. While this example is simple, the concept can be extended with many levels of overlap and priority.

Managing a ternary CAM with overlapping entries is more complex than managing a binary CAM, because the winning entry priority must be kept in sync with the application's needs, even as the CAM is updated during operation. A CAM is rarely initialized once and then left alone for the remainder of system operation. Its contents are modified periodically as network traffic changes. Let's say that the ZIP code CAM was initialized as follows in consecutive entries: 1121X, 11234, 1123X, 112XX. Where would a new special-case entry 11235 be placed? It would have to precede the 1123X entry for it to match before 1123X. Therefore, the system would have to temporarily move CAM entries to insert 11235 into the correct entry. If there is enough free space in the CAM, the system could initialize it and reserve free entries in between valid entries. But, sooner or later, the CAM will likely become congested in a local area, requiring it to be reorganized. How the data is arranged and how the CAM is reorganized will affect system performance, because it is likely that the CAM will have to be temporarily paused in its search function until the reorganization is complete. Solutions to this pause include a multibank CAM architecture whereby the system reorganizes the lookup table in an inactive bank and then quickly swaps inactive and active banks.

A CAM often does not associate general data bits with each entry, because the main purpose of a CAM is to match tags, not to store large quantities of data. It is therefore common to couple a CAM with an external SRAM that actually holds the data of interest and that can be arbitrarily expanded according to application requirements as shown in Fig. 8.19. In this example, the CAM contains

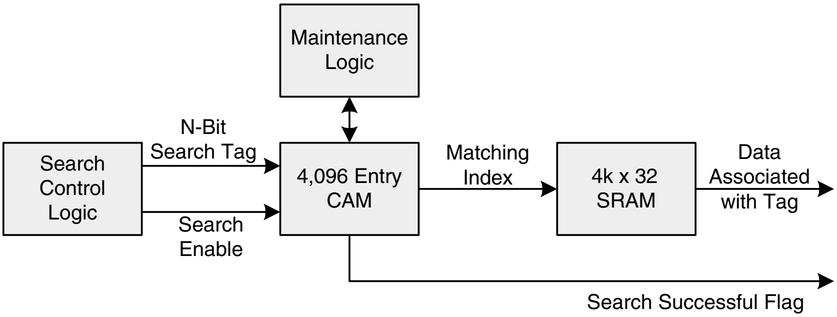


FIGURE 8.19 CAM augmentation with external SRAM.

4,096 entries and returns a 12-bit index when a tag has been successfully matched. This index serves as the address of an SRAM that has a 32-bit data path as required by the application.

When combined with conventional memory and some control logic, a CAM subsystem is sometimes referred to as a *search engine*. A search engine is differentiated from a stand-alone CAM by being capable of semi-autonomous lookups on behalf of another entity such as data processing logic in either hardware or software. A search engine's control logic can be as simple as accepting a search tag and then returning data along with a success flag. It can get more complex to include specific table maintenance functions so that CAM overhead operations are completely offloaded from the data processing logic. Search engines are especially useful when interfacing with special-purpose *network processor* devices. These processors run software to parse packets and make decisions about how each packet should be handled in the system. The tag lookup function is offloaded to a search engine when there is not enough time for a software algorithm to search a large table.

This page intentionally left blank.

CHAPTER 9
Networking

Data communications is an essential component of every digital system. Some systems realize com-

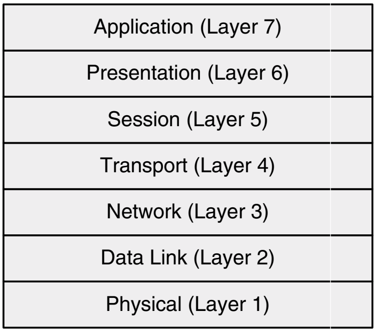


FIGURE 9.1 OSI seven-layer model.

may segment an OSI layer into multiple sublayers. The consistency of definitions decreases as one moves up the stack, because of functional protocol variations.

Layer one, the *physical layer*, comprises the electromechanical characteristics of the medium used to convey bits of information. The use of twisted pair cable, the amplitude of 1s and 0s, and associated connectors and transducers are examples of that which is specified in the physical layer. Channel coding, how the bits are represented on the physical medium, is usually classified as part of the physical layer.

The *data link layer*, layer two, encompasses the control logic and frame formatting that enables data to be injected into the network's physical layer and retrieved at the destination node. Layer-two functions are usually handled by a *media access controller* (MAC), a hardware device that contains all of the logic necessary to gain access to the network medium, properly format and transmit a frame, and properly detect and process an incoming frame. Network frame formats specify data link layer characteristics. Link level error detection mechanisms such as checksums and CRCs (more on these later) are generated and verified by the MAC. Node addresses, called *MAC addresses* in Ethernet networks, are layer-two constructs that uniquely identify individual nodes. Layer-two functions are usually handled in hardware, because they are repetitive, high-frequency, and time-critical operations. The data link layer is closely tied to the topology of the network because of its handling of access control functions and unique node addresses. Network *switches* operate at layer two by knowing which node address is connected to which port and then directing traffic to the relevant port. If port 20 of a switch is connected to node 87, all frames that enter the switch destined for node 87 will be sent out port 20. Because it is necessary to maintain unique layer-two addresses, they are generally not under the control of the user but rather are configured by the manufacturer. In the case of Ethernet, each manufacturer of equipment licenses an arbitrary range of MAC addresses from the IEEE and then assigns them one at a time as products roll off the assembly lines.

9.2 PROTOCOL LAYERS THREE AND FOUR

More flexible communications are possible when a protocol is not tied too closely to network topology or even the type of network accomplishing the exchange of information. The *network layer*, layer three, enables nodes to establish end-to-end connections without strict knowledge of the network topology. Layer-three packets are encapsulated within the payload of a layer-two frame. The packets typically contain their own header, payload, and sometimes a trailer as well. Perhaps the most common example of a layer-three protocol is *Internet Protocol* (IP). IP packets consist of a header and payload. Included within the header are 32-bit layer-three destination and source *IP addresses*. A separate set of network addresses can be implemented at layer three that is orthogonal to layer-two addresses. This gives network nodes two different addresses: one at layer three and one at layer two. For a simple network, this may appear to be redundant and inefficient. Yet modern networking protocols must support complex topologies that span buildings and continents, often with a mix of data links connecting many smaller subnetworks that may cover a single office or floor of a building. The benefit of layer-three addressing and communication is that traffic can be carried on a variety of underlying communications interfaces and not require the end points to know the exact characteristics of each interface.

Network *routers* operate at layer three by separating the many subnetworks that make up a larger network and only passing traffic that must travel between the subnetworks. Network addresses are typically broken into *subnets* that correlate to physically distinct portions of the network. A router has multiple ports, each of which is connected to a different subnetwork that is represented by a range of network addresses. A frame entering a router port will not be sent to another particular port on that router unless its network address matches a subnet configuration on that particular port. Strictly speaking, this separation could be performed by layer-two addressing, but the practical reality is that layer-two addresses are often not under the user's control (e.g., Ethernet) and therefore cannot be organized in a meaningful way. In contrast, layer-three addresses are soft properties of each network installation and are not tied to a particular type of network medium.

Layer-three functions are performed by both hardware and software according to the specific implementation and context. Layer-three packets are usually first generated by software but then manipulated by hardware as they flow through the network. A typical router processes layer-three packets in hardware so that it does not fall behind the flow of traffic and cause a bottleneck.

The bottom three layers cumulatively move data from one place to another but sometimes do not have the ability to actually guarantee that the data arrived intact. Layers one and two are collectively responsible for moving properly formatted frames onto the network medium and then recovering those in transit. The network layer adds some addressing flexibility on top of this basic function. A true end-to-end guarantee of data delivery is missing from certain lower-level protocols (e.g., Ethernet and IP) because of the complexity that this guarantee adds.

The *transport layer*, layer four, is responsible for ensuring end-to-end communication between software services running on each node. Transport layer complexity varies according to the demands of the application. Many applications are written with the simplifying assumption that once data is passed to the transport layer for transmission, it is guaranteed to arrive at the destination. *Transmission control protocol* (TCP) is one of the most common layer-four protocols, because it is used to guarantee the delivery of data across an unreliable IP network. When communicating via TCP, an application can simply transfer the desired information and then move on to new tasks. TCP is termed a *stateful* protocol, because it retains information about packets after they are sent until their successful arrival has been acknowledged. TCP operates using a sliding data transmission window shown in Fig. 9.2 and overlays a 32-bit range of indices onto the data that is being sent. Pointers are referenced into this 32-bit range to track data as it is transmitted and received.

The basic idea behind TCP is that the transmitter retains a copy of data that has already been sent until it receives an acknowledgement that the data was properly received at the other end. If an ac-

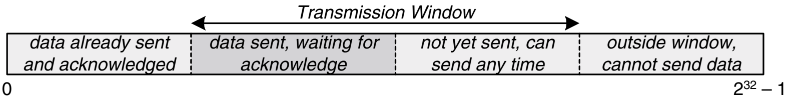
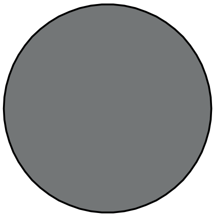

Transmission Window
data already sent and acknowledged
data sent, waiting for acknowledge
not yet sent, can send any time
outside window, cannot send data
0
$2^{32} - 1$

FIGURE 9.2 TCP transmission window.

plemented in network driver software. However, on special-purpose platforms where high bandwidth is critical, many layer-three and layer-four functions are accelerated by hardware. How these trade-offs are made depends on the exact type of networking scheme being implemented.

9.3 PHYSICAL MEDIA

Most wired networking schemes use high-speed unidirectional serial data channels as their physical communication medium. A pair of unidirectional channels is commonly used to provide bidirec-



to bounce off the core/cladding boundary as shown in Fig. 9.3b, thereby trapping the light over very long distances.

Light is injected into the core using either an LED or laser, depending on the required quality of the signal. A laser can generate light that is coherent, meaning that its photons are at the same frequency and phase. Injecting coherent light into a fiber optic cable reduces the distortion that accumulates over distance as photons of different frequency travel through the medium at slightly different velocities. Noncoherent photons that are emitted simultaneously as part of a signal pulse will arrive at the destination spread slightly apart in time. This spreading makes reconstructing the signal more difficult at very high frequencies, because signal edges are distorted.

Even when *coherent* light is used, photons can take multiple paths in the core as they bounce off the core/cladding boundary at different angles. These multiple propagation *modes* cause distortion over distance. To deal with this phenomenon, two types of fiber optic cable are commonly used: *single-mode* and *multimode*. Single-mode fiber contains a very thin core of approximately 8 to 10 μm in diameter that constrains light to a single propagation mode, thereby reducing distortion. Multimode fiber contains a larger core, typically 62.5 μm, that allows for multiple propagation modes and hence increased distortion. Single-mode fiber is more expensive than multimode and is used in longer-distance and higher-bandwidth applications as necessary.

Fiber optic cabling is more expensive than copper wire, and the handling of optical connections is more costly and complex as compared to copper. Splicing a fiber optic cable requires special equipment to ensure a clean cut and low-loss junction between two separate cables. The best splice is obtained by actually fusing two cables together to form a seamless connection. This is substantially more involved than splicing a copper cable, which can be done with fairly simple tools. Fiber optic connectors are sensitive to dirt and other contaminants that can attenuate the photons' energy as they pass through. Additionally, fine abrasive particles can scratch the glass faces of optical interfaces, causing permanent damage. Once properly installed and sealed, however, fiber optic cable can actually be more rugged than copper cables because of its insensitivity to oxidation that degrades copper wiring over time. Aside from bandwidth issues, these environmental benefits have resulted in infrastructures such as cable TV being partially reinstalled with fiber to cut long-term maintenance costs.

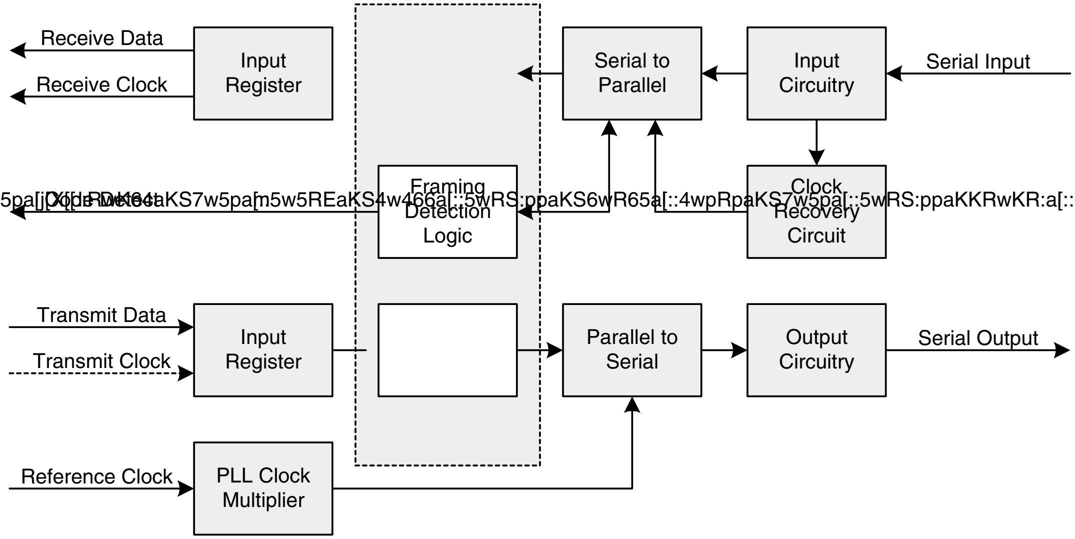
9.4 CHANNEL CODING

High-speed serial data channels require the basic functionality of a UART, albeit at very high speed, to convert back and forth between serial and parallel data paths. Unlike a UART that typically functions at kilobits or a few megabits per second, specialized transceiver ICs called *serializer/deserial-*

izers, or *serdes* for short, are manufactured that handle serial rates of multiple gigabits per second. Serdes vendors include AMCC, Conexant, Intel, PMC-Sierra, Texas Instruments, and Vitesse. To simplify system design, a serdes accepts a lower-frequency reference clock that is perhaps 1/10 or 1/20 the bit frequency of the serial medium. Parallel data is usually transmitted to the serdes at this reference frequency. For example, the raw bit rate of gigabit Ethernet (IEEE 802.3z) is 1.25 Gbps, but a typical serdes accepts a 125-MHz reference clock and 10 bits per cycle of transmit data. The reference clock is internally multiplied using a phase locked loop (PLL) to achieve the final bit rate. A general serdes block diagram is shown in Fig. 9.4. An optional transmit clock is shown separately from the reference clock, because some devices support these dual clocks. The benefits of a dual-clock scheme are that a very stable reference clock can be driven by a high-accuracy source separately from a somewhat noisier source-synchronous transmit clock generated by the data processing logic. This eases the clock jitter requirements on data processing logic.

A clock recovery circuit in the receiver portion extracts a bit clock from the serial data stream that is transmitted without a separate clock. This recovery is possible, because the serial data stream is normally coded with an algorithm that guarantees a certain proportion of state transitions regardless of the actual data being transferred. Such coding can be performed within the serdes or by external data processing logic. Channel coding is necessary for more than clock recovery. Analog circuits in the signal path, notably transducer and amplifier elements, require a relatively balanced data stream to function optimally. In circuit analysis terms, they work best when the data stream has an average DC value of 0. This is achieved with a data stream that contains an equal number of 1s and 0s over short spans of time. If a 1 is represented as a positive voltage and a 0 is represented as a negative voltage of equal magnitude, equal numbers of 1s and 0s balance out to an average voltage of 0 over time.

Fortunately, it is possible to encode an arbitrary data stream such that the coded version contains an average DC value of 0, and that data can be restored to its original form with an appropriate decoding circuit. One fairly simple method of encoding data is through a scrambling polynomial im-



plemented with a *linear feedback shift register*—a shift register with feedback generated by a set of exclusive-OR gates. The placement of the XOR feedback terms is mathematically defined by a binary polynomial. Figure 9.5 shows scrambling logic used to encode and decode eight-bit data words using the function F(X) = X^7+ X^4 + 1. The mathematical theory behind such polynomials is based on *Galois fields*, discovered by Evariste Galois, a nineteenth century French mathematician. XOR gates are placed at each bit position specified by the polynomial exponents, and their outputs feed back to the shift register input to scramble and feed forward to the output to descramble.

This type of scrambling should not be confused with more sophisticated security and data protection algorithms. Data scrambled in this manner is done so for purposes of randomizing the bits on the communications channel to achieve an average DC value of 0. Polynomial scrambling works fairly well and is relatively easy to implement, but such schemes are subject to undesired cases in which the application of select repetitive data patterns can cause an imbalance in the number of 1s and 0s, thereby reducing the benefit of scrambling. The probability of settling into such cases is low, making scrambling a suitable coding mechanism for certain data links.

While shown schematically as a serial process, these algorithms can be converted to parallel logic by accumulating successive XOR operations over eight bits shifted through the polynomial register. In cases when the coding logic lies outside of the serdes in custom logic, it is necessary to convert this serial process into a parallel one, because data coming from the serdes will be in parallel form at a corresponding clock frequency. Working out the logic for eight bits at a time allows processing one byte per clock cycle. The serial to parallel algorithm conversion can be done over any number of bits that is relevant to a particular application. This process is conceptually easy, but it is rather tedious to actually work out the necessary logic.

A table can be formed to keep track of the polynomial code vector, C[6:0], and the output vector, Q[7:0], as functions of the input vector, D[7:0]. Table 9.1 shows the state of C and Q, assuming that the least-significant bit (LSB) is transmitted first, during each of eight successive cycles by listing terms that are XORed together.

The final column, D[7], and the bottom row, Q, indicate the final state of the code and output vectors, respectively. The code vector terms can be simplified, because some iterative XOR feedback terms cancel each other out as a result of the identity that $A \oplus A = 0$. Q[7:0] can be taken directly from the table because there are no duplicate XOR terms. The simplified code vector, C[6:0], is shown in Table 9.2.

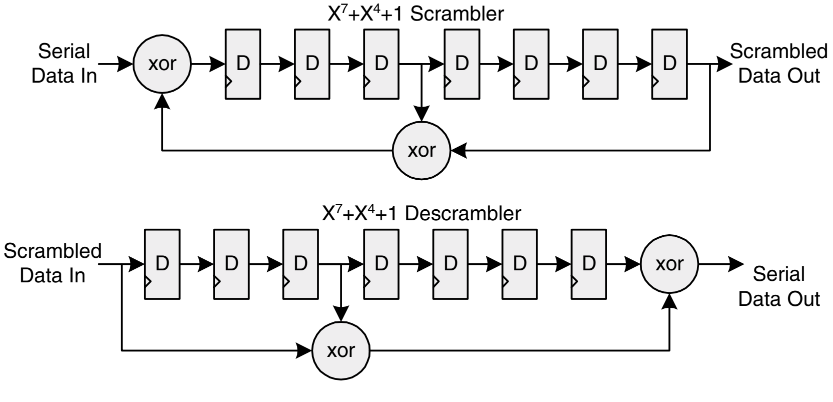


FIGURE 9.5 Eight-bit scrambling/descrambling logic.

TABLE 9.3 Descrambling Logic Code and Output Vector Logic

Code Vector Bits	Shift Logic	Output Vector Bits	XOR Logic
–	–	D7	D0 D7 D4
C6	D7	D6	C6 D4antp—dB M

Data that has been encoded must be decoded before framing information can be extracted. Therefore, the serdes' receiving shift register must begin by performing a simple serial-to-parallel conversion until framing information can be extracted after decoding. Prior to the detection of framing information, the output of the parallel shift register will be arbitrary data at an arbitrary alignment, because there is no knowledge of where individual bytes of words begin and end in the continuous data stream. Once a framing sequence has been detected, the shift register can be "snapped" into correct alignment, and its output will be properly formatted whole bytes or words.

When reconstructing decoded data, the desired byte alignment will likely span two consecutive bytes as they come straight from the descrambling logic. The most significant bits of a descrambled byte logically follow the LSB of the next descrambled byte because of the order in which bits are shifted through the scrambler and descrambler. Therefore, when data arrives misaligned at the receiver/descrambler, bytes are reassembled by selecting the correct bits from the most significant bits (MSB) of descrambled byte N and the LSB of descrambled byte N + 1 as shown in Fig. 9.6.

Framing information is not conveyed by the scrambled coding and must therefore be extracted at a higher level. For this reason, certain serdes components that are used in scrambled coding systems

TABLE 9.5 5B6B Sub-block Encoding

Input Character	Binary Value HGF	Encoded Value fghj	
		Positive Disparity	Negative Disparity
D/Kxx.0	000	1011	0100
Dxx.1	001	1001	
Kxx.1	001	0110	1001
Dxx.2	010	0101	
Kxx.2	010	1010	0101
D/Kxx.3	011	1100	0011
D/Kxx.4	100	1101	0010
Dxx.5	101	1010	
Kxx.5	101	0101	1010
Dxx.6	110	0110	
Kxx.6	110	1001	0110
Dxx.7	111	1110 (0111)	0001 (1000)
Kxx.7	111	0111	1000

ous character to the current 5B6B code. If the 5B6B code is neutral, CRD´ will reflect the disparity of the generated 5B6B code. Otherwise, the lookup table will attempt to choose a code that balances out the CRD. If the character maps to a neutral code, the CRD will be passed through. The 3B4B table not only performs a simple mapping of the Y sub-block, but it also handles the alternate encoding of the special cases mentioned previously. The final CRD from this table is stored for use in the next character encoding.

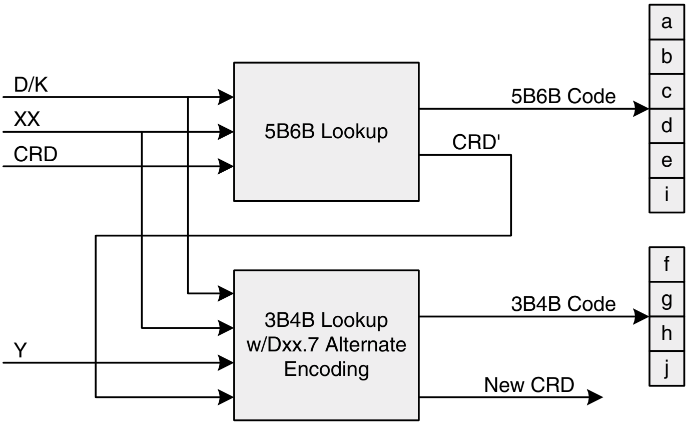


FIGURE 9.8 8B10B encoding logic.

9.7 CHECKSUM

A checksum is a summation of a set of data and can be an arbitrary width, usually 8, 16, or 32 bits. Once a set of data has been summed, the checksum is sent along with the frame so that the sum can be verified at the receiver. The receiver can calculate its own checksum value by summing the relevant data and then compare its result against the frame's checksum. Alternatively, many checksum schemes enable the checksum value itself to be summed along with the data set, with a final result of zero indicating verification and nonzero indicating an error.

Perhaps the most common checksum scheme uses one's complement binary arithmetic to calculate the sum of a data set. One's complement addition involves calculating a normal two's complement sum of two values and then adding the carry bit back into the result. This constrains the running sum to the desired bit width, a necessary feature when summing hundreds or thousands of bytes where an unconstrained sum can be quite large. It is guaranteed that, when the carry bit is added back into the original result, a second carry will not be generated. For example, adding 0xFF and 0xFF yields 0x1FE. When the carry bit is added back into the eight-bit sum, 0xFF is the final one's complement result.

An interesting consequence of one's complement math is that the eight-bit values 0x00 and 0xFF (or 0xFFFF for a 16-bit value) are numerically equivalent. Consider what would happen if 1 is added to either value. In the first case, 0x00 + 0x01 = 0x01, is the obvious result. In the second case, 0xFF + 0x01 = 0x100 = 0x01, where the carry bit is added back into the eight-bit sum to yield the same final result, 0x01. A brief example of calculating a 16-bit checksum is shown in Table 9.6 to aid in understanding the one's complement checksum.

TABLE 9.6 Sixteen-Bit One's Complement Checksum

Data Value	Sum	Carry	Running Checksum
Initialize checksum to zero			0x0000
0x1020	0x1020	0	0x1020
0xFFF0	0x1010	1	0x1011
0xAD00	0xBD11	0	0xBD11
0x6098	0x1DA9	1	0x1DAA
0x701E	0x8DC8	0	0x8DC8

The logic to perform a one's complement checksum calculation can take the form of a normal two's complement adder whereby the carry bit is fed back in the next clock cycle to adjust the sum. As shown in Fig. 9.9, this forms a pipelined checksum calculator where the latency is two cycles. To begin the calculation, the accumulator and carry bit are reset to 0. Each time a word is to be summed, the multiplexer is switched from 0 to select the word. The adder has no qualifying logic and adds its two inputs, whose sum is repeatedly loaded into the accumulator and carry bits on each rising clock edge. After the last word has been summed, the control logic should wait an extra cycle, during which the multiplexer is selected to 0 to allow the most recent carry bit to be incorporated into the sum. It is guaranteed that a nonzero carry bit will not propagate into another nonzero carry bit after summing the accumulator with 0. This configuration works well in many situations, because two's complement adders are supported by many available logic implementation technologies. In high-speed designs, logic paths with adders in them can prove difficult to meet timing. As shown in Fig. 9.9, an optional input register can be added between the multiplexer and the adder to completely isolate the adder from the control logic. This modification improves timing at the expense of an added cycle of latency to the checksum calculation.

commonly called the *Header Error Check* (HEC) field, is defined by the polynomial $x^8 + x^2 + x + 1$. The HEC is implemented with an eight-bit *linear feedback shift register* (LFSR) as shown in Fig. 9.10. Bytes are shifted into the LFSR one bit at a time, starting with the MSB. Input data and the last CRC bit feed to the XOR gates that are located at the bit positions indicated by the defining polynomial. After each byte has been shifted in, a CRC value can be read out in parallel with the LSB and MSB at the positions shown. When a new CRC calculation begins, this CRC algorithm specifies that the CRC register be initialized to 0x00. Not all CRC algorithms start with a 0 value; some start with each bit set to 1.

The serial LFSR can be converted into a set of parallel equations to enable practical implementation of the HEC on byte-wide interfaces. The general method of deriving the parallel equations is the same as done previously for the scrambling polynomial. Unfortunately, this is a very tedious process that is prone to human error. As CRC algorithms increase in size and complexity, the task can get lengthy. LFSRs may be converted manually or with the help of a computer program or spreadsheet. Table 9.7 lists the XOR terms for the eight-bit HEC algorithm wherein a whole byte is clocked through each cycle. Each CRC bit is referred to as Cn, where n = [7:0]. Once the equations are simplified, matching pairs of CRC and data input bits are found grouped together. Therefore, the convention Xn is adopted where Xn = Cn XOR Dn to simplify notation. Similar Boolean equations can be derived for arbitrary cases where fractions of a byte (e.g., four bits) are clocked through each cycle, or where multiple bytes are clocked through in the case of a wider data path.

TABLE 9.7 Simplified Parallel HEC Logic

CRC Bits	XOR Logic
C0	X0 X6 X7
C1	X0 X1 X6
C2	X0 X1 X2 X6
C3	X1 X2 X3 X7
C4	X2 X3 X4
C5	X3 X4 X5
C6	X4 X5 X6
C7	X5 X6 X7

When a new HEC calculation is to be started, the CRC state bits are reset to 0. Each byte is then
locked through the parallel logic at the rate of one byte per cycle. Following the final data byte, the
EC is XORed with 0x55 to yield a final result. An arbitrary number of bytes can be clocked
ough, and the CRC value will change each cycle. The one exception to this is the case of leading
Because the HEC specifies a reset state of 0, passing 0x00 data through the CRC logic will not
t in a nonzero value. However, once a nonzero value has been clocked through, the LFSR will
ain a nonzero value in the presence of a stream of 0s. This property makes the HEC nonideal
cking arbitrary strings of leading 0s, and it is a reason why other CRC schemes begin with a
reset value. Table 9.8 shows an example of passing four nonzero data bytes through the par-
logic and then XORing with 0x55 to determine a final CRC value.

xamples of HEC Calculation

t	HEC Value
	0x00
	0x77
	0xAC
	0xD4
	0xF9
	0xAC

it polynomial appropriately called *CRC-16*. Its polynomial is
entation is shown in Fig. 9.11. As with the HEC, a CRC-16
ion. Because the CRC-16 is two bytes wide, its common
he data path is 8 or 16 bits wide. Of course, wider data
of more complex logic. Table 9.9 lists the CRC-16
cycle.
bit shuffling to conform to industry conven-
to the CRC algorithm, it is important for
circuit can properly exchange CRC
to LSB, the CRC-16 shifts in data
h-byte, bits [15:8], of a 16-bit
order in which bytes are
must be flipped before
b ic to the task, and

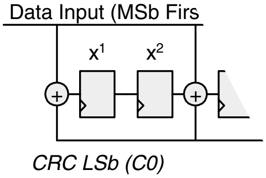


FIGURE 9.11 CRC-16 LFSR.

the CRC-16, parallel CRC-32 logic is commonly derived for data paths of one, two, or four bytes in width. A difference between the CRC-32 and those CRC schemes already presented is that the CRC32's state bits are initialized to 1s rather than 0s, and the final result is inverted before being used. Table 9.11 lists the CRC-32 XOR terms for handling one, two, or four bytes per cycle.

As noted, the CRC-32 state bits are initialized with 1s before calculation begins on a new data set. Words are byte-swapped and bit-flipped according to the same scheme as done for the CRC-16. When the last data word has been clocked through the parallel logic, the CRC-32 state bits are inverted to yield the final calculated value. Table 9.12 shows a step-by-step example of calculating a CRC-32 32 bits at a time using the same 32-bit data set, 0x4D41524B, as before.

CRC algorithms can be performed in software, and often are when cost savings is more important ws Hfx9H©GD`C©UE88wx*C9—D————C9—D©Gww©ted'©Gww—C*±o÷f[st x—©fsGGGC*±o

TABLE 9.11 CRC-32 Parallel Logic

CRC Bits	32-Bit XOR Logic	16-Bit XOR Logic	8-Bit XOR Logic
C0	X0 X6 X9 X10 X12 X16 X24 X25 X26 X28 X29 X30 X31	C16 C22 C25 C26 C28 D0 D6 D9 D10 D12	C24 C30 D0 D6
C1	X0 X1 X6 X7 X9 X11 X12 X13 X16 X17 X24 X27 X28	C16 C17 C22 C23 C25 C27 C28 C29 D0 D1 D6 D7 D9 D11 D12 D13	C24 C25 C30 C31 D0 D1 D6 D7
C2	X0 X1 X2 X6 X7 X8 X9 X13 X14 X16 X17 X18 X24 X26 X30 X31	C16 C17 C18 C22 C23 C24 C25 C29 C30 D0 D1 D2 D6 D7 D8 D9 D13 D14	C24 C25 C26 C30 C31 D0 D1 D2 D6 D7
C3	X1 X2 X3 X7 X8 X9 X10 X14 X15 X17 X18 X19 X25 X27 X31	C17 C18 C19 C23 C24 C25 C26 C30 C31 D1 D2 D3 D7 D8 D9 D10 D14 D15	C25 C26 C27 C31 D1 D2 D3 D7
C4	X0 X2 X3 X4 X6 X8 X11 X12 X15 X18 X19 X20 X24 X25 X29 X30 X31	C16 C18 C19 C20 C22 C24 C27 C28 C31 D0 D2 D3 D4 D6 D8 D11 D12 D15	C24 C26 C27 C28 C30 D0 D2 D3 D4 D6
C5	X0 X1 X3 X4 X5 X6 X7 X10 X13 X19 X20 X21 X24 X28 X29	C16 C17 C19 C20 C21 C22 C23 C26 C29 D0 D1 D3 D4 D5 D6 D7 D10 D13	C24 C25 C27 C28 C29 C30 C31 D0 D1 D3 D4 D5 D6 D7
C6	X1 X2 X4 X5 X6 X7 X8 X11 X14 X20 X21 X22 X25 X29 X30	C17 C18 C20 C21 C22 C23 C24 C27 C30 D1 D2 D4 D5 D6 D7 D8 D11 D14	C25 C26 C28 C29 C30 C31 D1 D2 D4 D5 D6 D7
C7	X0 X2 X3 X5 X7 X8 X10 X15 X16 X21 X22 X23 X24 X25 X28 X29	C16 C18 C19 C21 C23 C24 C26 C31 D0 D2 D3 D5 D7 D8 D10 D15	C24 C26 C27 C29 C31 D0 D2 D3 D5 D7
C8	X0 X1 X3 X4 X8 X10 X11 X12 X17 X22 X23 X28 X31	C16 C17 C19 C20 C24 C26 C27 C28 D0 D1 D3 D4 D8 D10 D11 D12	X0 C24 C25 C27 C28 D1 D3 D4
C9	X1 X2 X4 X5 X9 X11 X12 X13 X18 X23 X24 X29	C17 C18 C20 C21 C25 C27 C28 C29 D1 D2 D4 D5 D9 D11 D12 D13	X1 C25 C26 C28 C29 D2 D4 D5
C10	X0 X2 X3 X5 X9 X13 X14 X16 X19 X26 X28 X29 X31	C16 C18 C19 C21 C25 C29 C30 D0 D2 D3 D5 D9 D13 D14	X2 C24 C26 C27 C29 D0 D3 D5
C11	X0 X1 X3 X4 X9 X12 X14 X15 X16 X17 X20 X24 X25 X26 X27 X28 X31	C16 C17 C19 C20 C25 C28 C30 C31 D0 D1 D3 D4 D9 D12 D14 D15	X3 C24 C25 C27 C28 D0 D1 D4
C12	X0 X1 X2 X4 X5 X6 X9 X12 X13 X15 X17 X18 X21 X24 X27 X30 X31	C16 C17 C18 C20 C21 C22 C25 C28 C29 C31 D0 D1 D2 D4 D5 D6 D9 D12 D13 D15	X4 C24 C25 C26 C28 C29 C30 D0 D1 D2 D5 D6
C13	X1 X2 X3 X5 X6 X7 X10 X13 X14 X16 X18 X19 X22 X25 X28 X31	C17 C18 C19 C21 C22 C23 C26 C29 C30 D1 D2 D3 D5 D6 D7 D10 D13 D14	X5 C25 C26 C27 C29 C30 C31 D1 D2 D3 D6 D7
C14	X2 X3 X4 X6 X7 X8 X11 X14 X15 X17 X19 X20 X23 X26 X29	C18 C19 C20 C22 C23 C24 C27 C30 C31 D2 D3 D4 D6 D7 D8 D11 D14 D15	X6 C26 C27 C28 C30 C31 D2 D3 D4 D7
C15	X3 X4 X5 X7 X8 X9 X12 X15 X16 X18 X20 X21 X24 X27 X30	C19 C20 C21 C23 C24 C25 C28 C31 D3 D4 D5 D7 D8 D9 D12 D15	X7 C27 C28 C29 C31 D3 D4 D5
C16	X0 X4 X5 X8 X12 X13 X17 X19 X21 X22 X24 X26 X29 X30	X0 C16 C20 C21 C24 C28 C29 D4 D5 D8 D12 D13	C8 C24 C28 C29 D0 D4 D5
C17	X1 X5 X6 X9 X13 X14 X18 X20 X22 X23 X25 X27 X30 X31	X1 C17 C21 C22 C25 C29 C30 D5 D6 D9 D13 D14	C9 C25 C29 C30 D1 D5 D6

TABLE 9.12 Step-by-Step CRC-32 Calculation

Operation	Data
Initialize CRC-32 state bits	0xFFFFFFFF
Word to be calculated	0x4D41524B
Reorder bytes to end high-byte first after bit-flipping	0x4B52414D
Flip bits for LSB-first transmission of high-byte then low-byte	0xB2824AD2
Clock word through XOR logic	0x5C0778F5
Flip bits of CRC	0xAF1EE03A
Optionally swap bytes of CRC for final result	0x3AE01EAF
Invert CRC state bits when input stream is completed	0xC51FE150

3 Mbps), including 10-, 100-, and 1,000-Mbps varieties. Ten-gigabit Ethernet is just now beginning to emerge. Ethernet originally ran over single shared segments of coaxial cabling, but most modern installations use twisted pair wiring in a physical star configuration. The familiar standards for Ethernet over twisted pair are 10BASE-T, 100BASE-T, and 1000BASE-T.

There is a whole family of Ethernet and related standards defined by the IEEE under the 802 LAN/MAN (local area network/metropolitan area network) Standards Committee. More specifically, the 802.3 CSMA/CD (carrier sense, multiple access, collision detect) Working Group defines Ethernet in its many forms. The 802.3 Ethernet frame format is shown in Table 9.13. A seven-byte preamble and a start of frame delimiter (collectively, a preamble) precede the main portion of the frame, which includes the header, payload, and trailer. The purpose of the preamble is to assist receivers in recognizing that a new frame is being sent so that it is ready to capture the main portion of the frame when it propagates through the wire. Not including the preamble, a traditional Ethernet frame ranges from 64 to 1,518 bytes. Two 48-bit Ethernet, or MAC, addresses are located at the start of the header: a destination address followed by a source address. The MSB of the address, bit 47, defines whether the address is unicast (0) or multicast (1). A unicast address defines a single source or destination node. A multicast address defines a group of destination nodes. The remaining address bits are broken into a 23-bit vendor block code (bits 46 through 24) and a 24-bit vendor-specific unique identifier (23 through 0). Manufacturers of Ethernet equipment license a unique vendor block code from the IEEE and then are responsible for assigning unique MAC addresses for all of their products. Each vendor block code covers 16 million (2^{24}) unique addresses.

Following the addresses is a length/type field that has two possible uses, for historical reasons. Prior to IEEE standardization, Xerox got together with Intel and Digital Equipment Corporation to agree on a standard Ethernet frame called *DIX*. DIX defines a type field that uniquely identifies the type of payload (e.g., IP) to enable easier parsing of the frame. When the IEEE first standardized Ethernet, it decided to implement a length field in place of a type field to more easily handle situations wherein payloads were less than the minimum 46 bytes allowed by the standard. This bifurcation of Ethernet caused interoperability problems. Years later, in 1997, the IEEE changed the field to be a combined length/type field. Values up to 1500 are considered lengths, and 1501 and above are considered types. Most Ethernet implementations use the original DIX-type field scheme. The IEEE has standardized a variety of type values to identify IP and certain other protocol extensions. Payloads with fewer than 46 bytes must be *padded* with extra data to meet the minimum frame size. The

TABLE 9.13 IEEE 802.3 Ethernet Frame Format

Field	Bytes	Fixed Value
Preamble	7	0x55
Start of frame delimiter	1	0xD5
Destination address	6	No
Source address	6	No
Length/type	2	No
Payload data	46–1500	No
Frame check sequence (CRC)	4	No

resolution of how many real data bytes are actually in an Ethernet frame is typically handled by higher-level protocols, such as IP, that contain their own length fields.

Modern Ethernet frames can be longer than 1,518 bytes for a couple of reasons. First, the IEEE has defined various data fields that can be thought of as extensions to the traditional Ethernet header. These include VLAN (virtual LAN) and MPLS (multiprotocol label switching) tags. Each of these extensions provides additional addressing and routing information for more advanced networking devices and adds length to the frame. Second, the industry began supporting *jumbo frames* in the late 1990s to extend an Ethernet frame to 9 kB. The advantage of a jumbo frame is that the same amount of data can be transferred with fewer individual frames, reducing overhead. Jumbo frame support is not universal, however. Older Ethernet equipment most likely will not handle such frames.

The frame check sequence is a 32-bit CRC that is computed across the entire main portion of the frame from the destination address through the last payload byte.

Because of its original topology as a shared bus, Ethernet employs a fairly simple yet effective arbitration mechanism to share access to the physical medium. This scheme is collision detection with random back-off, which was discussed earlier. Ethernet is referred to as CSMA/CD because of its access sharing mechanism. Frame size plays a role in the operation of CSMA/CD. A minimum frame size is necessary to ensure that, for a given physical network size, all nodes are capable of properly detecting a collision in time to take the correct action. Electrical signals propagate through copper wire at a finite velocity. Therefore, if two nodes at opposite ends of a bus begin transmitting at the same time, it will take a finite time for each to recognize a collision. Once a frame is successfully in progress, all other nodes must wait for that frame to end before they can transmit. A maximum frame size limits the time that a single node can occupy the shared network. Additionally, a maximum frame size limits the buffer size within Ethernet MAC logic. In the 1970s and early 1980s, the cost of memory was so high as to justify relatively small maximum frame sizes. Today, this is not a significant concern in most products, hence the emergence of jumbo frames.

Even when Ethernet networks are deployed in physical star configurations, they are often connected to hubs that electrically merge the star segments into a single, logically shared medium. A traditional Ethernet network is half-duplex, because only one frame can be in transit at any instant in time. Hubs are the least expensive way to connect several computers via Ethernet, because they do little more than merge star segments into a bus. Bus topologies present a fixed pool of bandwidth that must be shared by all nodes on that bus. As the number of nodes on a network increases, the traffic load is likely to increase as well. Therefore, there is a practical limit on the size of a bussed Ethernet network. *Bridges* were developed to mitigate Ethernet congestion problems by connecting multiple independent bus segments. A bridge operates at layer two using MAC addresses and builds a database of which addresses are on which side of the bridge. Only traffic that must cross the bridge to another segment is actually passed to the relevant segment. Otherwise, traffic can remain local on a single Ethernet segment without causing congestion on other segments. Bridging is illustrated in Fig. 9.13. Nodes 1 and 7 can simultaneously send data within their local segments. Later, node 4 can send data across the bridge to node 8, during which both network segments are burdened with the single transfer. The simplicity of this approach is that node 4 does not have any knowledge that node 8 is on a different segment. Crossing between Ethernet segments is handled transparently by the bridge.

Layer-two switches take bridging a step farther by providing many independent Ethernet ports that can exchange frames simultaneously without necessarily interfering with the traffic of other ports. As long as multiple ports are not trying to send data to the same destination port, those ports can all send data to different ports as if there existed many separate dedicated connections within the switch. This is known as *packet switching*: instantaneous connections between ports are made and then broken on a packet-by-packet basis. If two or more ports try to send data to the same port at the same time, one port will be allowed to transmit, while the others will not. Ethernet was developed to be a simple and inexpensive technology. Therefore, rather than providing special logic to handle such congestion issues, it was assumed that the network would generally have sufficient bandwidth to serve the application. During brief periods of high demand where not all data could be reliably delivered, it was assumed that higher-level protocols (e.g., TCP/IP) would handle such special cases in software, thereby saving money in reducing hardware complexity at the expense of throughput. Traditional Ethernet switches simply drop frames when congestion arises. In the case of switch congestion wherein multiple ports are sending data to a single port, all but one of those source ports may have their frames discarded. In reality, most switches contain a small amount of buffering that can temporarily hold a small number of frames that would otherwise be discarded as a result of congestion. However, these buffers do not prevent frame drops when congestion rises above a certain threshold. This behavior underscores the utility of layer-four protocols such as TCP.

Each switch port can conceivably be connected to a separate bused Ethernet segment and provide bridging functions on a broader scale than older bridges with only two ports. Switching has transformed network architecture as the cost of hardware has dropped over the years. It is common to find central computing resources such as file servers and printers with dedicated switch ports as

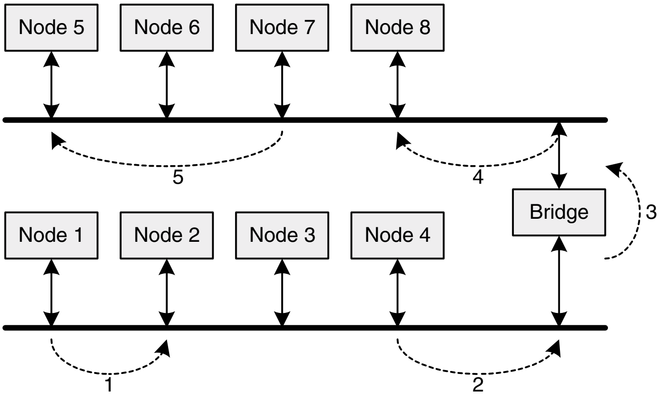


FIGURE 9.13 Ethernet bridging.

shown in Fig. 9.14. Other switch ports may connect to less-expensive hubs (some switches have built-in hubs to create a compact, integrated system). This reduces congestion by placing the most actively used nodes onto *dedicated media*, thereby eliminating collisions and increasing overall system bandwidth. Assuming 100BASE-T Ethernet segments, each file server and printer has a dedicated 100-Mbps data link into the switch. All arbitration and congestion control can be handled within the switch. If two file servers were placed onto the same *shared media* Ethernet segment, they would have to share the 100-Mbps bandwidth of a single segment and would likely experience collisions as many nodes tried to exchange data with them.

Switches and dedicated media transform Ethernet into a full-duplex-capable data link by providing separate transmit and receive signal paths between the switch and a node. Full-duplex operation is a subset of half-duplex operation, because the frame formatting is identical, but the CSMA/CD algorithm is not necessary for dedicated media applications. Some MACs may be designed for dedicated media only and can be made simpler without the necessity of media sharing logic. However, high-volume MACs may be designed into a variety of applications, requiring them to support both half- and full-duplex operation.

Ethernet ports on switches typically are capable of operating at multiple data rates to enable greater compatibility with other devices. To ease interoperability between MACs that can run at different speeds, 10/100/1000BASE-T has a mechanism called *autonegotiation* that enables two MACs to automatically determine their highest common data rate. The MACs ultimately must run at the same speed to properly exchange frames, but the initialization process of configuring the link to operate at the greatest common speed has been standardized and automated at the MAC layer. Autonegotiation works by each MAC exchanging a 16-bit message at a speed that is compatible with the slowest port type (10BASE-T). Each MAC advertises its capabilities, including speed and half/full-duplex support. The MACs may then select the greatest mutually supported link attributes. Autonegotiation is supported only for point-to-point links and is therefore most commonly observed between a switch and whatever entity is connected to it (e.g., another switch, a node, etc.).

```
module my_logic (
  A, B, C, Y
);

input A, B, C;
output Y;

wire and1_out, and2_out, notA;

and_gate u_and1 (
  .in1 (A),
  .in2 (B),
  .out (and1_out)
);

not_gate u_not (
  .in  (A),
  .out (notA)
);

and_gate u_and2 (
  .in1 (notA),
  .in2 (C),
  .out (and2_out)
);

or_gate u_or (
  .in1 (and1_out),
  .in2 (and2_out),
  .out (Y)
);

endmodule
```

FIGURE 10.1 Verilog gate/instance level design.

ues at discrete events, as will be soon discussed. When ports are defined, they are assumed to be wires unless declared otherwise. An output port can be declared as a type other than wire.

Being that this example is a gate/instance-level design, all logic is represented by instantiating other modules that have been defined elsewhere. A module is instantiated by invoking its name and following it with an instance name. Here, the common convention of preceding the name with "u_" is used, and multiple instances of the same module type are differentiated by following the name with a number. Individual ports for each module instance are explicitly connected by referencing the port name prefixed with a period and then placing the connecting variable in parentheses. Ports can be implicitly connected by listing only connecting variables in the order in which a module's ports are defined. This is generally considered poor practice, because it is prone to mistakes and is difficult to read.

HDL's textual representation of logic is converted into actual gates through a process called *logic synthesis*. A synthesis program parses the HDL code and generates a *netlist* that contains a detailed list of low-level logic gates and their interconnecting nets, or wires. Synthesis is usually done with a specific implementation target in mind, because each implementation technology differs in the logic primitives that it provides as basic building blocks. The primitive library for an ASIC will differ from that of a PLD, for example. Once synthesis is performed, the netlist can be transformed into a working chip and, hence, a working product.

A key benefit of HDL design methodology is the ability to thoroughly simulate logic before committing a netlist to a real chip. Because HDL is a programming methodology, it can be arbitrarily manipulated in a software simulation environment. The simulator allows a *test bench* to be written in either the HDL or another language (e.g., C/C++) that is responsible for creating stimulus to be applied to the logic modules. Widely used simulators include Cadence's NC-Sim, Model Technology's ModelSim, and Synopsys' VCS and Scirocco. A distinction is made between synthesizable and nonsynthesizable code when writing RTL and test benches. Synthesizable code is that which represents the logic to be implemented in some type of chip. Nonsynthesizable code is used to implement the test bench and usually contains constructs specifically designed for simulation that cannot be converted into real logic through synthesis.

An example of a test bench for the preceding Verilog module might consist of three number generators that apply pseudo-random test stimulus to the three input ports. Automatic verification of the logic would be possible by having the test bench independently compute the function $Y = A\&B + \overline{A}\&C$ and then check the result against the module's output. Such simulation, or verification, techniques can be used to root out the great majority of bugs in a complex design. This is a tremendous feature, because fixing bugs after an ASIC has been fabricated is costly and time consuming. Even in cases in which a PLD is used, it is usually faster to isolate and fix a bug in simulation than in the laboratory. In simulation, there is immediate access to all internal nodes of the design. In the lab, such access may prove quite difficult to achieve.

Verilog and VHDL both support simulation constructs that facilitate writing effective test benches. It is important to realize that these constructs are usually nonsynthesizable (e.g., a random number generator) and that they should be used only for writing test code rather than actual logic.

Gate/instance-level coding is quite useful and is used to varying degrees in almost every design, but the real power of HDL lies at the RTL and behavioral levels. Except in rare circumstances where absolute control over gates is required, instance-level coding is used mainly to connect different modules together. Most logic is written in RTL and behavioral constructs which are often treated together, hence the reason that synthesizable HDL code is often called RTL. Expressing logic in RTL frees the engineer from having to break everything down into individual gates and transfers this responsibility onto the synthesis software. The result is a dramatic increase in productivity and maintainability, because logical representations become concise. The example in Fig. 10.1 can be rewritten in Verilog RTL in multiple styles as shown in Fig. 10.2.

Each of these three styles has its advantages, each is substantially more concise and readable than the gate/instance-level version, and the styles can be freely mixed within the same module according to the engineer's preference. Style number 1 is a *continuous assignment* and makes use of the default wire data type for the output port. A wire is applicable here, because it is implicitly connecting two entities: the logic function and the output port. Continuous assignments are useful in certain cases, because they are concise, but they cannot get too complex without becoming unwieldy.

Style number 2 uses the always block, a keyword that tells the synthesis and simulation tools to perform the specified operations whenever a variable in its *sensitivity list* changes. The sensitivity list defines the variables that are relevant to the always block. If not all relevant variables are included in this list, incorrect results may occur. Always blocks are one of Verilog's fundamental constructs. A design may contain numerous always blocks, each of which contains logic functions that are activated when a variable in the sensitivity list changes state. A combinatorial always block should normally include all of its input variables in the sensitivity list. Failure to do so can lead to unexpected simulation results, because the always block will not be activated if a variable changes state and is not in the sensitivity list.

Style number 3 also uses the always block, but it uses a logical if…else construct in place of Boolean expression. Such logical representations are often preferable so that an engineer can concentrate on the functionality of the logic rather than deriving and simplifying Boolean algebra.

```
// synchronous reset

always @(posedge CLK)
begin
  if (RESET) // RESET evaluated only at CLK rising edge
    Q <= 1´b0;
  else
    Q <= D;
end

// asynchronous reset

always @(posedge CLK or posedge RESET)
begin
  if (RESET) // RESET evaluated whenever it goes active
    Q <= 1´b0;
  else
    Q <= D;
end
```

FIGURE 10.3 Verilog RTL flip-flop inference.

the non-blocking assignment does not take effect until after the current simulation time unit. This is analogous to the behavior of a real flop wherein the output does not transition until a finite time has elapsed from its triggering event. Under certain circumstances, either type of assignment will yield the same result in both simulation and synthesis. In other situations, the results will differ, as illustrated in Fig. 10.4.

In the first case, regs Q1 and Q2 are tracked at two different instants in time. First, their current states are maintained as they were just prior to the clock edge for the purpose of using their values in subsequent assignments. Second, their new states are assigned as dictated by the RTL. When Q2 is assigned, it takes the previous value of Q1, not the new value of Q1, which is D. Two flops are inferred.

In the second case, variables Q1 and Q2 are tracked at a single instant in time. Q1 is assigned the value of variable D, and then Q2 is assigned the new value of variable Q1. Q1 has become a temporary placeholder and has no real effect on its own. Therefore, only a single flop, Q2, is inferred.

Utilizing HDL to design logic requires software tools more complex than just pencil and paper. However, the benefits quickly accumulate for designs of even moderate complexity. The digital

```
// Non-blocking assignments: two flops inferred

always @(posedge CLK)
begin
  Q1 <= D;
  Q2 <= Q1;
end

// Blocking assignments: one flop inferred

always @(posedge CLK)
begin
  Q1 = D;
  Q2 = Q1;
end
```

FIGURE 10.4 Verilog blocking vs. non-blocking assignment.

installed into the CS1_ slot. A jumper can then be installed that causes RomSel to be asserted. When the system is turned on, RomSel=1 causes the ROM module to become the boot ROM, and new software can be loaded into CS0_ ROM.

The remainder of the address space is sparsely populated. Occupied memory regions are spread out to reduce the complexity of the decoding logic by virtue of requiring fewer address bits. If the UART were located immediately after the SRAM, the logic would have to consider the state of A[23:16] rather than just A[23:20]. The fifth and final used memory region is reserved for internal control and status registers. This decoding logic can be written in Verilog as shown in Fig. 10.5.

The address decoding logic is written here in behavioral form with a *case* construct. Case statements enable actions to be associated with individual states of a causal variable. Note that the chip select outputs are declared as regs even though they are not flops, because they are assigned in an always block instead of in a continuous assignment. Prior to the case statement, all of the always

```
module GlueLogic (
  Addr,
  RomSel,
  CS0_,
  CS1_,
  CS2_,
  CS3_
);

input  [23:20] Addr;
input          RomSel;
output         CS0_, CS1_, CS2_, CS3_;

reg            CS0_, CS1_, CS2_, CS3_;
reg            IntSel;

always @(Addr or RomSel)
begin
  CS0_   = 1´b1;  // establish default values to simplify case
  CS1_   = 1´b1;  // statement and prevent formation of latches
  CS2_   = 1´b1;
  CS3_   = 1´b1;
  IntSel = 1´b0;

  case (Addr[23:20])
    4´b0000 : begin
                CS0_ = RomSel;
                CS1_ = !RomSel;
              end
    4´b0001 : begin
                CS0_ = !RomSel;
                CS1_ = RomSel;
              End
    4´b0010 : CS2_   = 1´b0;
    4´b0011 : CS3_   = 1´b0;
    4´b0100 : IntSel = 1´b1;
  endcase
end

endmodule
```

FIGURE 10.5 Address decoding logic.

```
always @(CS1_ or Rd_)
begin
  if (!CS1_ && !Rd_)
    DataBufDir = 1’b0;  // drive CPU bus when ROM selected for read
  else
    DataBufDir = 1’b1;  // otherwise, always drive data to ROM
end
```

FIGURE 10.7 Data buffer control logic.

on an alarm. The opening door can be detected using a switch connected to an input signal. When the CPU reads the status of this signal, it can determine whether the switch is open or closed. An alarm can be turned on when the CPU sets an output signal that enables an alarm circuit. Control and status registers must be implemented to enable the CPU to read and write I/O signals. In our continuing example, we assume an eight-bit data bus coming from the CPU and the need for eight input signals and eight output signals. Implementing registers varies according to whether the CPU bus is synchronous or asynchronous. Some older microprocessors use asynchronous buses requiring latches to be formed within the support logic. Figure 10.8 shows the implementation of two registers using the previously decoded IntSel signal in both synchronous and asynchronous styles. Again, the proper declarations for ports and variables are assumed.

An added level of address decoding is required here to ensure that the two registers are not accessed simultaneously. The register logic consists of two basic sections: the write logic and read logic. The write logic (required only for the control register that drives output signals) transfers the contents of the CPU data bus to the internal register when the register is addressed and the write enable is active. The ControlRegSel signal is implemented in a case statement but can be implemented in a variety of ways. More select signals will be added in coming examples. The asynchronous write logic infers a latch, because not all permutations of input qualifiers are represented by assignments. If Reset_ is high and the control register is not being selected for a write, there is no specified action. Therefore, memory is implied and, in the absence of a causal clock, a latch is inferred. The synchronous write logic is almost identical, but it references a clock that causes a flop inference. Reset is implemented to provide a known initial state. This is a good idea so that external logic that is driven by the control register can be safely designed with the assumption that operations begin at a known state. The known state is usually inactive so that peripherals do not start operating before the CPU finishes booting and can disable them.

The read logic consists of two sections: the output multiplexer and the output buffer control. The output multiplexer simply selects one of the available registers for reading. It is not necessary to qualify the multiplexer with any other logic, because a read will not actually take place unless the output buffer control logic sends the data to the CPU. Rather than preventing a latch in ReadData by assigning it a default value before the case construct, the Verilog keyword *default* is used as the final case enumeration to specify default operation. Either solution will work—it is a matter of preference and style over which to use in a given situation. Both read-only and writable registers are included in the read multiplexer logic. Strictly speaking, it is not mandatory to have writable register contents readable by the CPU, but this is a very good practice. Years ago, when logic was very expensive, it was not uncommon to find write-only registers. However, there is a substantial drawback to this approach: you can never be sure what the contents of the register are if you fail to keep track of the exact data that has already been written!

Implementing bidirectional signals in Verilog can be done with a continuous assignment that selects between driving an active variable or a high-impedance value, Z. The asynchronous read logic is very simple: whenever the internal registers are selected and read enable is active, the tri-state buffer is enabled, and the output of the multiplexer is driven onto the CPU data bus. At all other

```
always @(Addr[3:0] or StatusInput[7:0] or ControlReg[7:0] or IntSel)
begin
  case (Addr[3:0]) // read multiplexer
    4´h0    : ReadData[7:0] = StatusInput[7:0]; // external input pins
    4´h1    : ReadData[7:0] = ControlReg[7:0];
    default : ReadData[7:0] = 8´h0; // alternate means to prevent latch
  endcase

  ControlRegSel = 1´b0;  // default inactive value

  case (Addr[3:0]) // select signal only needed for writeable registers
    4´h1 : ControlRegSel = IntSel;
  endcase
end

// Option #1A: asynchronous read logic

assign CpuData[7:0] = (IntSel && !Rd_) ? ReadData[7:0] : 8´bz;

// Option #1B: synchronous read logic

always @(posedge CpuClk)
begin
  if (!Reset_)  // synchronous reset
    CpuDataOE <= 1´b0;
    // no need to reset ReadDataReg and possibly save some logic
  else begin
    CpuDataOE        <= IntSel && !Rd_; // all outputs are registered
    ReadDataReg[7:0] <= ReadData[7:0];
  end
end

assign CpuData[7:0] = CpuDataOE ? ReadDataReg[7:0] : 8´bz;

// Option #2A:asynchronous write logic

always @(ControlRegSel or CpuData[7:0] or Wr_ or Reset_)
begin
  if (!Reset_)
    ControlReg[7:0] = 8´h0;  // reset state is cleared
  else if (ControlRegSel && !Wr_)
    ControlReg[7:0] = CpuData[7:0];
  // missing else forces memory element: intentional latch!
end

// Option #2B: synchronous write logic

always @(posedge CpuClk)
begin
  if (!Reset_)  // synchronous reset
    ControlReg[7:0] <= 8´h0;
  else if (ControlRegSel && !Wr_)
    ControlReg[7:0] <= CpuData[7:0];
end
```

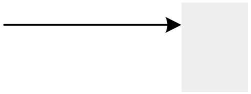
FIGURE 10.8 Control/status register logic.

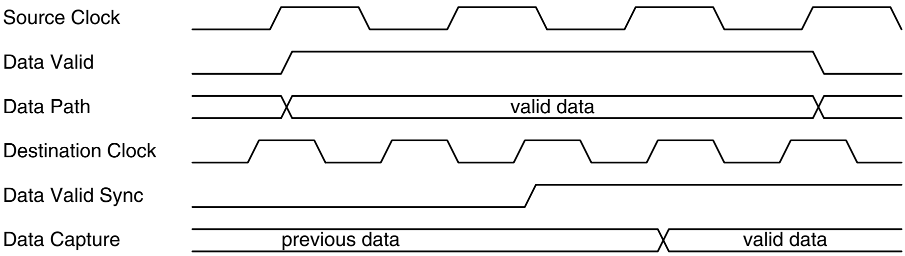
times, the data bus is held in a high-impedance state. This works as expected, because the value Z can be overridden by another assignment. In simulation, the other assignment may come from a test bench that emulates the CPU's operation. In synthesis, the software properly recognizes this arrangement as inferring a tri-state bus. The continuous assignment takes advantage of Verilog's conditional operator, ? : , which serves an if…else function. When the logical expression before the question mark is true, the value before the colon is used. Otherwise, the value after the colon is used. A bidirectional port is declared using the Verilog *inout* keyword in place of *input* or *output*. The synchronous version of the read logic is very similar to the asynchronous version, except that the outputs are first registered before being used in the tri-state assignment.

Interrupt control registers are implemented by support logic when the number of total interrupt

output may not transition until the next clock cycle. Whether the input signal meets or misses the first flop's timing window, an extra cycle of latency is added by the second synchronizing flop.

For the synchronizing circuit to function properly, the asynchronous input must remain active for a sufficient time to be detected by the destination clock domain. If the destination clock has a 20-ns period, pulses shorter than 20 ns will have a low probability of being detected. It is best to guarantee that the pulse is active for several destination clock periods. When the two clock frequencies in a clock domain crossing are known, it is possible to determine a safe minimum pulse width. There are also situations in which one or both clock frequencies are unknown or variable. A circuit may need





and read ports. If a FIFO is used to carry 8-bit data from a 100-MHz clock to a 50-MHz clock, it is clear that the overall data rate cannot exceed the slower read rate, 50 MBps, without overflowing the FIFO and losing data. If the situation is reversed, there is still a 50-MB overall maximum data rate, and the 100-MHz read logic must inherently handle gaps in its data path, which has a 100-MB bandwidth. As long as the read and write data rates are matched over time, the dual-clock FIFO will never overflow or underflow and provides an efficient synchronization mechanism.

10.4 FINITE STATE MACHINES

Finite state machines (FSMs) are powerful design elements used to implement algorithms in hardware. A loose analogy is that an FSM is to logic design what programming is to software design. An FSM consists of a *state vector*, a set of registers, with associated logic that advances the state each clock cycle depending on external input and the current state. In this respect, an FSM is analogous to a set of software instructions that are sequenced via a microprocessor's program counter. Each state can be designed to take a different arbitrary action and branch to any other state in the same conceptual way that software branches to different program sections as its algorithm dictates. If a problem can be decomposed into a set of deterministic logical steps, that algorithm can be implemented with an FSM.

A counter is a simple example of an FSM, though its actions are very limited. Each state simply advances to the next state on each clock cycle. There is no conditional branching in a typical counter. FSMs are often represented graphically before being committed to RTL. Figure 10.15 shows a bubble diagram representation of a two-bit counter. Each state is represented by its own bubble, and arcs show the conditions that cause one state to lead to other states. An unlabeled arc is taken to mean un-

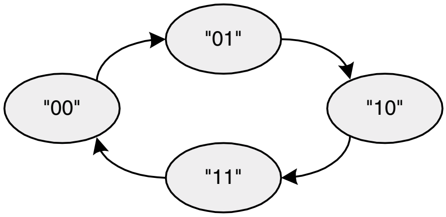
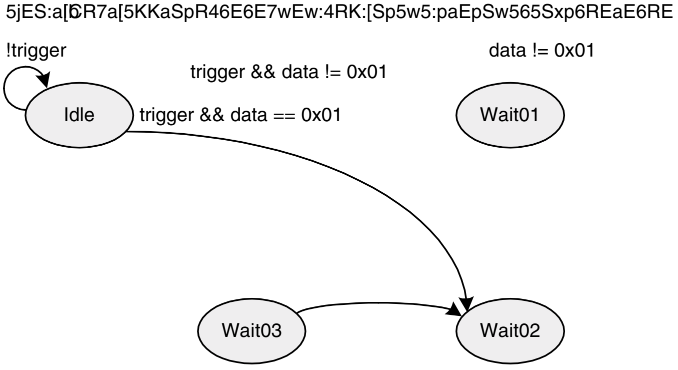


FIGURE 10.15 Two-bit counter bubble diagram.

conditional or the default if no other conditional arcs are valid. Because this is a simple counter, each state has one unconditional arc that leads to the next state.

To illustrate basic FSM concepts, consider a stream of bytes being driven onto a data interface and a task to detect a certain pattern of data within that stream when a trigger signal is asserted. That pattern is the consecutive set of data 0x01, 0x02, 0x03, and 0x04. When this consecutive pattern has been detected, an output detection flag should be set to indicate a match. The logic should never miss detecting this pattern, no matter what data precedes or follows it. After some consideration, the bubble diagram in Fig. 10.16 can be developed. It is common to name states with useful names such as "Idle" rather than binary numbers for readability. FSMs often sit in an idle or quiescent state while they wait for a triggering event that starts their execution.

The FSM waits in the Idle state until trigger is asserted. If the data value at this point is already 0x01, the FSM skips directly to the Wait02 state. Otherwise, it branches to Wait01, where it remains indefinitely until the value 0x01 is observed and it branches to Wait02. There are three possible arcs coming from Wait02. If the value 0x02 is observed, the FSM can advance to the third matching state, Wait03. If 0x01 is observed again, the FSM remains in Wait02, because 0x02 is the next value in the sequence following 0x01. If neither of these values is observed, the FSM branches back to Wait01 to start over again. Wait03 is similar, although there is no arc to remain in the same state. If the first sequence value, 0x01, is observed, the next state is Wait02. If Wait03 does succeed in immediately observing 0x03, the FSM can advance to the final matching state, Wait04. If Wait04 immediately observes 0x04, a match is declared by asserting the Match output flag, and the FSM completes its function by returning to Idle. Otherwise, like in Wait03, the FSM branches back to Wait01 or Wait02.



FSMs are formally classified into one of two types: *Moore* and *Mealy*. A Moore FSM is one wherein the output signals are functions of only the current state vector. A Mealy FSM is one in which the output signals are functions of both the current state as well as the inputs to the FSM. The pattern-matching FSM is a Mealy FSM, because Match is asserted when in state Wait04 and the input data is equal to 0x04. It could be converted into a Moore FSM by adding an additional state, perhaps called Matched, that would be inserted in the arc between Wait04 and Idle. The benefit of doing this would be to reduce logic complexity, because the output signals are functions of only the state vector rather than the inputs as well. However, this comes at the cost of adding an extra state. As FSMs get fairly complex, it may be too cumbersome to rigidly conform to a Moore design.

An FSM is typically coded in HDL using two blocks: a sequential (clocked) state vector block and a combinatorial state and output assignment block. The pattern-matching FSM can be written in Verilog as shown in Fig. 10.17. Rather than calling out numerical state values directly in the RTL, constants are defined to make the code more readable. Verilog supports the *`define* construct that associates a literal text string with a text identifier. The output, Match, is explicitly registered before leaving the module so that downstream logic timing will not be slowed down by incurring the penalty of the FSM logic. Registering the signal means that the output is that of a flop directly, rather than through an arbitrary number of logic gates that create additional timing delays.

In reading the FSM code, it can be seen that the state vector is three bits wide, because there are five states in total. However, 2^3 equals 8, indicating that three of the possible state vector values are invalid. Although none of these invalid values should ever arise, system reliability is increased by inserting the *default* clause into the case statement. This assignment ensures that any invalid state will result in the FSM returning to Idle. Without a default clause, any invalid state would not cause any action to be taken, and the FSM would remain in the same state indefinitely. It is good practice to insert default clauses into case statements when writing FSMs to guard against a hung condition. It is admittedly unlikely that the state vector will assume an invalid value, but if a momentary glitch were to happen, the design would benefit from a small amount of logic to restore the FSM to a valid state.

10.5 FSM BUS CONTROL

FSMs are well suited to managing arbitrarily complex bus interfaces. Microprocessors, memory devices, and I/O controllers can have interfaces with multiple states to handle arbitration between multiple requestors, wait states, and multiword transfers (e.g., a burst SDRAM). Each of these bus states can be represented by an FSM state, thereby breaking a seemingly complex problem into quite manageable pieces.

A simple example of CPU bus control is an asynchronous CPU interface with request/acknowledge signaling in which the FSM runs on a different clock from that of the CPU itself. Asynchronous CPU buses tend to implement four-corner handshaking to prevent the CPU and peripheral logic from getting out of step with one another. Two asynchronous control signals, chip select (CS_) and

```
module PatternMatch (
  Clk,
  Reset_,
  Trigger,
  Data,
  Match
);

input         Clk, Reset_, Trigger;
input   [7:0] Data;
output        Match;

reg           Match, NextMatch;
reg     [2:0] State, NextState;

`define IDLE    3´h0
`define WAIT01  3´h1
`define WAIT02  3´h2
`define WAIT03  3´h3
`define WAIT04  3´h4

// sequential state vector block

always @(posedge Clk)
begin
  if (!Reset_)
    State[2:0] <= `IDLE;
  else
    State[2:0] <= NextState[2:0];
end

// register output of FSM
// could be combined with above state vector block if desired

always @(posedge Clk)
begin
  if (!Reset_)
    Match <= 1´b0;
  else
    Match <= NextMatch;
end

// FSM combinatorial logic

always @(State[2:0] or Trigger or Data[7:0])
begin
  NextState[2:0] = State[2:0]; // default values prevent latches
  NextMatch      = 1´b0;

  case (State[2:0])

    `IDLE :
       if (Trigger && (Data[7:0] == 8´h01))
         NextState[2:0] = `WAIT02;
       else if (Trigger)  // data != 0x01
         NextState[2:0] = `WAIT01;
```

FIGURE 10.17 Pattern-matching FSM logic.

secondary transaction on behalf of the microprocessor and then transition to a state that waits until the secondary transaction completes before acknowledging the microprocessor.

It may appear that writing a formal FSM when the state vector is a single flop is too cumbersome.

```
reg  State, NextState;
reg  CS_Input_, CS_Sync_, CpuAck_;

`define IDLE  1´h0
`define ACK   1´h1

always @(posedge Clk)
begin
  if (!Reset_)
    State <= `IDLE;
  else
    State <= NextState;
end

// FSM support logic: synchronization and registered output

always @(posedge Clk)
begin
  if (!Reset_) begin
    CS_Input_ <= 1´b1; // active low signals reset to high
    CS_Sync_  <= 1´b1;
    CpuAck_   <= 1´b1;
  end
  else begin
    CS_Input_ <= CS_;       // first synchronizer stage
    CS_Sync_  <= CS_Input_; // second synchronizer stage

    if (SetAck)
      CpuAck_ <= 1´b1;
    else if (ClrAck)
      CpuAck_ <= 1´b0;
  end
end

// FSM logic assumes supporting logic:
//
// CpuDataOE enables tristate output for reads
// WriteEnable enables writes to registers decoded from address inputs

always @(State or CS_Sync_ or Rd_ or Wr_)
begin
  NextState = State; // default values prevent latches

  ClrAck      = 1´b0;
  CpuDataOE   = 1´b0;
  SetAck      = 1´b0;
  WriteEnable = 1´b0;

  case (State)

    `IDLE :
       if (!CS_Sync_) begin
         NextState   = `ACK;
         ClrAck      = 1´b1;
         CpuDataOE   = !Rd_;
         WriteEnable = !Wr_;
       end

    `ACK :
       if (CS_Sync_) begin  // wait for CS_ deassertion
         NextState   = `IDLE;
         SetAck      = 1´b1;
       end

  endcase
end
```

FIGURE 10.19 Asynchronous bus slave logic.

sus programmable logic) provide more or less flexibility in how flops and logic gates are placed together. In many cases, custom logic (e.g., an ASIC) is more efficient with a binary encoded FSM, whereas programmable logic performs better with moderately sized one-hot FSMs.

An FSM can be explicitly written as one-hot, but most leading HDL synthesis tools (e.g., Cadence BuildGates, Exemplar Leonardo Spectrum, Synplicity Synplify, or Synopsys Design Compiler and FPGA Express) have options to automatically evaluate a binary encoded FSM in either its

10.7 PIPELINING

Pipelining is a significant method of improving FSM timing in a way similar to normal logic. A complex FSM contains a large set of inputs that feeds the next state logic. As the number of branch variables across the entire FSM increases, long timing paths quickly develop. It is advantageous to precompute many branch conditions and reduce a condition to a binary variable: true or false. In doing so, the branch variables are moved away from the FSM logic and behind a pipeline flop. The logic delay penalty of evaluating the branch condition is hidden by the pipeline flop. This simplifies the FSM logic because it has to evaluate only the true/false condition result, which is a function of one variable, instead of the whole branch condition, which can be a function of many variables.

An example of using pipelining to improve FSM timing is whereby an FSM is counting events and waits in a particular state until the event has occurred N times, at which point the FSM branches to a new state. In this situation, a counter lies off to the side of the FSM, and the FSM asserts a signal to increment the counter each time a particular event occurs. Without pipelining, the FSM would compare the counter bits against the constant, N, as a branch condition. If the counter is eight bits wide, there is a function of eight variables plus any other qualifiers. If the counter is 16 bits wide, at least 16 variables are present. Many such Boolean equations in the same FSM can lead to timing problems.

Pipelining can be implemented by performing the comparison outside of the FSM with the true/false result being stored in a flop that is directly evaluated by the FSM, thereby reducing the evaluation to a function of only one variable. The trick to making this work properly is that the latency of the pipelined comparison needs to be built into both the comparison logic and the FSM itself.

In a counter situation without pipelining, the FSM increments the counter when an event occurs and simultaneously evaluates the current count value to determine if N events have occurred. Therefore, the FSM would have to make its comparison against N – 1 instead of against N if it wanted to react at the proper time. When the counter is at N – 1 and the event occurs, the FSM knows that this event is the Nth event in the sequence.

Inserting a pipeline flop into the counter comparison logic adds a cycle of latency to the comparison. The FSM cannot simply look ahead an additional cycle to N – 2, because it cannot tell the future: it does not ordinarily know if an event will occur in the next cycle. This problem can be addressed in the pipelined comparison logic. The comparison logic must be triggered at N – 2 so that the FSM can observe the asserted signal on the following cycle when the counter is at N – 1, which enables the FSM to operate as before to recognize the Nth event. Because an ultimate comparison to N – 1 is desired, the pipelined logic can evaluate the expression, "If count equals N – 2 and increment is true, then count will equal N – 1 in the next cycle." In the next cycle, the FSM will have the knowledge that count equals N – 1 from a single flop, and it can qualify this in the same way it did without the pipelining with the end same result. Verilog code fragments to implement this scheme are shown in Fig. 10.21, where the FSM completes its task when 100 events have been observed.

Pipelining should be carefully considered when an FSM is evaluating long bit vectors if it is believed that proper timing will be difficult to achieve. Counter comparisons to constants are not the only candidates for pipelining. Valid pipelining candidates include counter comparisons to other registers and arithmetic embedded within branch conditions. Arithmetic expressions such as one shown in Fig. 10.22 can add long timing paths to an already complex FSM.

Adding two vectors and then comparing their sum to a third vector is a substantial quantity of logic. Such situations are not uncommon in an FSM that must negotiate complex algorithms. It would be highly advantageous to perform the arithmetic and comparison in a pipelined manner so that the FSM would have to evaluate only a single flop. The complexity involved in pipelining such an expression is highly dependent on the application's characteristics. Pipelining can get tricky when it becomes necessary to look ahead one cycle and deal with possible exceptions to the look-ahead

```
// pipelined pattern matching flops

always @(posedge Clk)
begin
  if (!Reset_) begin
 33).265 TD[(an$1dw8l183tching flops)-15( )]T-0.00ing6p 1383n$a$BI.0024 Tc 0 t7#s
```

When pipeline flops are inserted into a data path, care must be taken to apply a consistent delay to the entire data path. This example does not use the data path for any purpose other than pattern matching; therefore, there is no need to perform further operations on the data once it is checked for the relevant patterns. If, however, the logic performed pattern matching and then manipulated the data based on that matching (e.g., recognize the start of a data packet and then store the packet into a memory), it would be critical to keep the pipelined pattern-matching flops coincident with the data that they represent. Failure to do so would result in detecting the pattern late with respect to the data stream, thereby missing the packet's initial data. The data path can be delayed along with the pipelined logic by passing it through a register also. If the previous example did process the data, logic reading `DataPipe[7:0] <= Data[7:0]` could be placed into the same always block as the pattern matching logic. Subsequent references to Data[7:0] would be replaced by references to DataPipe[7:0]. This way, when Data[7:0] equals 0x01, Data01 will be set on the next cycle, and DataPipe[7:0] will be loaded with the value 0x01 on the next cycle as well.

This page intentionally left blank.

CHAPTER 11
Programmable Logic Devices

Programmable logic is the means stom
logic, whether that logic is a simp
is implemented with some type o
Boolean expressions each tim es
mable logic include rapid c te
port.
The widespread avail le l
design capabilities to m omp
nancial and staffing re m IC.
of sizes, operating v l but g
closely matched w ting tha
manufacturer ha ty and ra
Programma ces rapidl
vices with in speeds. Afte
basic types able, it is rec
manufact updated inform
Lattice, provide detailed
bundle re for reasonable pri

11.1 CUST RAMMABLE LOGIC

discrete 7400 ICs, custom logic is imp
ustom masks at a factory or programmed wi
n. Custom ICs, or *application specific integrated c*
cause, as with anything custom, there are fewer constrai
plemented. Because custom ICs are tailored for a specific app
gh clock speeds and relatively low unit prices. The disadvantages to cus
pensive development cycles and the inability to make quick logic changes. Custom
cycles are long, because a design must generally be frozen in a final state before much of the
layout and circuit design work can be completed. Engineering charges for designing a custom mask set (not including the logic design work) can range from $50,000 to well over $1 million, depending on the complexity. Once manufactured, the logic can't simply be altered, because the logic configuration is an inherent property of the custom design. If a bug is found, the time and money to alter the mask set can approach that of the initial design itself.

their cost differential. Likewise, raw packaging costs are likely to be comparable because of the maturation of stable packaging materials and technologies. The cost differential comes down to which solution requires the smaller die and how the overhead costs of manufacturing and distribution are amortized across the volume of chips shipped.

Die size is function of two design characteristics: how much logic is required and how many I/O pins are required. While the size of logic gates has decreased by orders of magnitude over time, the

once a microscopic fuse is blown, it cannot be restored. Today's devices typically rely on EEPROM technology and CMOS switches to enable nonvolatile reprogrammability. However, fuse-based terminology remains in use for historical reasons. The default configuration of a connection emulates an intact fuse, thereby connecting the input term to the AND gate input. When the connection is blown, or programmed, the AND input is disconnected from the input term and pulled high to effectively remove that input from the Boolean expression. Customization of a GAL's programmable AND gate is conceptually illustrated in Fig. 11.3.

With full programmability of the AND array, the OR connections can be hard wired. Each GAL device feeds a differing number of AND terms into the OR gates. If one or more of these AND terms

There are two common GAL devices, the 16V8 and the 22V10, although other variants exist as well. They contain eight and ten macrocells each, respectively. The 16V8 provides up to 10 dedicated inputs that feed the AND array, whereas the 22V10 provides 12 dedicated inputs. One of the 22V10's dedicated inputs also serves as a global clock for any flops that are enabled in the macrocells. Output enable logic in a 22V10 is evaluated independently for each macrocell via a dedicated AND term. The 16V8 is somewhat less flexible, because it cannot arbitrarily feed back all macrocell outputs depending on the device configuration. Additionally, when configured for registered mode where macrocell flops are usable, two dedicated input pins are lost to clock and output enable functions.

GALs are fairly low-density PLDs by modern standards, but their advantages of low cost and high speed are derived from their small size. Implementing logic in a GAL follows several basic steps. First, the logic is represented in either graphical schematic diagram or textual (HDL) form. This representation is converted into a netlist using a translation or synthesis tool. Finally, the netlist is fitted into the target device by mapping individual gate functions into the programmable AND array. Given the fixed AND/OR structure of a GAL, fitting software is designed to perform logic optimizations and translations to convert arbitrary Boolean expressions into sum-of-product expressions. The result of the fitting process is a programming image, also called a *fuse map*, that defines exactly which connections, or fuses, are to be programmed and which are to be left at their default state. The programming image also contains other information such as macrocell configuration and other device-specific attributes.

Modern PLD development software allows the back-end GAL synthesis and fitting process to proceed without manual intervention in most cases. The straightforward logic flow through the programmable AND array reduces the permutations of how a given Boolean expression can be implemented and results in very predictable logic fitting. An input signal propagates through the pin and pad structure directly into the AND array, passes through just two gates, and can then either feed a macrocell flop or drive directly out through an I/O pin. Logic elements within a GAL are close to each other as a result of the GAL's small size, which contributes to low internal propagation delays. These characteristics enable the GAL architecture to deliver very fast timing specifications, because signals follow deterministic paths with low propagation delays.

GALs are a logic implementation technology with very predictable capabilities. If the desired logic cannot fit within the GAL, there may not be much that can be done without optimizing the algorithm or partitioning the design across multiple devices. If the logic fits but does not meet timing, the logic must be optimized, or a faster device must be found. Because of the GAL's basic fitting process and architecture, there isn't the same opportunity of tweaking the device as can be done with more complex PLDs. This should not be construed as a lack of flexibility on the part of the GAL. Rather, the GAL does what it says it does, and it is up to the engineer to properly apply the technology to solve the problem at hand. It is the simplicity of the GAL architecture that is its greatest strength.

Lattice Semiconductor's GAL22LV10D-4 device features a worst-case input-to-output combinatorial propagation delay of just 4 ns.* This timing makes the part suitable for address decoding on fast microprocessor interfaces. The same 22V10 part features a 3-ns t_{CO} and up to 250-MHz operation. The t_{CO} specification is a pin-to-pin metric that includes the propagation delays of the clock through the input pin and the output signal through the output pin. Internally, the actual flop itself exhibits a faster t_{CO} that becomes relevant for internal logic feedback paths. Maximum clock frequency specifications are an interesting aspect of all PLDs and some consideration. These specifications are best-case numbers usually obtained with minimal logic configurations. They may define

* GAL22LV10D, 22LV10_04, Lattice Semiconductor, 2000, p. 7.

the highest toggle rate of the device's flops, but synchronous timing analysis dictates that there is more to f_{MAX} than the flop's t_{SU} and t_{CO}. Propagation delay of logic and connectivity between flops is of prime concern. The GAL architecture's deterministic and fast logic feedback paths reduces the added penalty of internal propagation delays. Lattice's GAL22LV10D features an internal clock-to-feedback delay of 2.5 ns, which is the combination of the actual flop's t_{CO} plus the propagation delay of the signal back through the AND/OR array. This feedback delay, when combined with the flop's 3-ns t_{SU}, yields a practical f_{MAX} of 182 MHz when dealing with most normal synchronous logic that contains feedback paths (e.g., a state machine).

11.3 CPLDS

Complex PLDs, or CPLDs, are the mainstream macrocell-based PLDs in the industry today, providing logic densities and capabilities well beyond those of a GAL device. GALs are flexible for their size because of the large programmable AND matrix that defines logical connections between inputs

Generic user I/O pins are bidirectional and can be configured as inputs, outputs, or both. This is in contrast to the dedicated power and test pins that are necessary for operation. There are as many potential user I/O pins as there are macrocells, although some CPLDs may be housed in packages that do not have enough physical pins to connect to all the chip's I/O sites. Such chips are intended for applications that are logic limited rather than pad limited.

Because the size of each logic block's AND array is fixed, the block has a fixed number of possible inputs. Vendor-supplied fitting software must determine which logical functions are placed into which blocks and how the switch matrix connects feedback paths and input pins to the appropriate block. The switch matrix does not grow linearly as more logic blocks are added. However, the impact of the switch matrix's growth is less than what would result with an ever expanding AND matrix. Each CPLD family provides a different number of switched input terms to each logic block.

The logic blocks share many characteristics with a GAL, as shown in Fig. 11.6, although additional flexibility is added in the form of product term sharing. Limiting the number of product terms in each logic block reduces device complexity and cost. Some vendors provide just five product terms per macrocell. To balance this limitation, which could impact a CPLD's usefulness, product term sharing resources enable one macrocell to borrow terms from neighboring macrocells. This borrowing usually comes at a small propagation delay penalty but provides necessary flexibility in handling complex Boolean expressions with many product terms. A logic block's macrocell contains a flip-flop and various configuration options such as polarity and clock control. As a result of their higher logic density, CPLDs contain multiple global clocks that individual macrocells can choose from, as well as the ability to generate clocks from the logic array itself.

Xilinx is a vendor of CPLD products and manufactures a family known as the XC9500. Logic blocks, or function blocks in Xilinx's terminology, each contain 18 macrocells, the outputs of which feed back into the switch matrix and drive I/O pins as well. XC9500 CPLDs contain multiples of 18 macrocells in densities from 36 to 288 macrocells. Each function block gets 54 input terms from the switch matrix. These input terms can be any combination of I/O pin inputs and feedback terms from other function blocks' macrocells.

Like a GAL, CPLD timing is very predictable because of the deterministic nature of the logic blocks' AND arrays and the input term switch matrix. Xilinx's XC9536XV-3 features a maximum pin-to-pin propagation delay of 3.5 ns and a t_{CO} of 2.5 ns.* Internal logic can run as fast as 277 MHz with feedback delays included, although complex Boolean expressions likely reduce this f_{MAX} because of product term sharing and feedback delays through multiple macrocells.

CPLD fitting software is typically provided by the silicon vendor, because the underlying silicon architectures are proprietary and not disclosed in sufficient detail for a third party to design the necessary algorithms. These tools accept a netlist from a schematic tool or HDL synthesizer and automatically divide the logic across macrocells and logic blocks. The fitting process is more complex

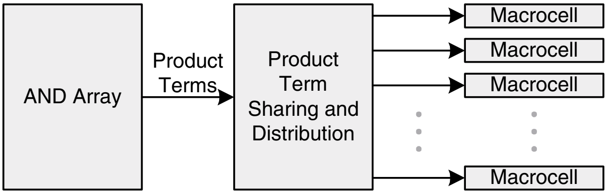


FIGURE 11.6 CPLD logic block.O8—CwA—OH—©CUA—ODUwA—OHAODUGHCU©A}UHG—fO8—CwA—OCw

* XC9536XV, DS053 (v2.2), Xilinx, August 2001, p. 4.

clock so that the causal edge observed by the I/O flops occurs at nearly the same time as when it enters the FPGA's clock input pin. PLLs and DLLs are discussed in more detail in a later chapter.

Additional circuitry enables some PLLs and DLLs to emit a clock that is related to the input frequency by a programmable ratio. The ability to multiply and divide clocks is a benefit to some system designs. An external board-level interface may run at a slower frequency to make circuit implementation easier, but it may be desired to run the internal FPGA logic as a faster multiple of that clock for processing performance reasons. Depending on the exact implementation, multiplication or division can assist with this scheme.

RAM blocks embedded within the logic cell array are a critical feature for many applications. FIFOs and small buffers figure prominently in a variety of data processing architectures. Without on-chip RAM, valuable I/O resources and speed penalties would be given up to use off-chip memory devices. To suit a wide range of applications, RAMs need to be highly configurable and flexible. A typical FPGA's RAM block is based on a certain bit density and can be used in arbitrary width/depth configurations as shown in Fig. 11.9 using the example of a 4-kb RAM block. Single- and dual-port modes are also very important. Many applications, including FIFOs, benefit from a dual-ported

[redacted]

three
provide co

More advanc
structures to achieve t
that specifically support DDR
as shown in Fig. 11.11. When configu
a pair of flops, and a multiplexer selects the
clock. A DDR interface runs externally to the FPGA o
Internally, the interface runs at double the external width on only
quency. Therefore, the I/O cell serves as a 2:1 multiplexer for outputs and a 1:
puts when operating in DDR mode.

This page intentionally left blank.

P • A • R • T • 3

ANALOG BASICS FOR DIGITAL SYSTEMS

This equation shows that a uniform loop current develops a voltage drop across one or more resistances in the circuit and that this total voltage must be offset by voltage sources. A loop equation can be written for Fig. 12.1, but special attention should be paid to the polarity of the voltage source versus the voltage drop through the resistor. The convention used to specify polarity does not change the final answer as long as the convention is applied consistently. Mistakes in loop analysis can arise from inconsistent representation of voltage polarities. In this case, the current is shown to circulate clockwise, so positive currents are clockwise currents. This means that the 50-Ω resistor will exhibit a voltage drop as the current moves through it from left to right. At the same time, the voltage source exhibits a voltage rise as the clockwise loop current passes through it. The voltage of the resistor and the voltage of the source are of opposite polarity as expressed in the following loop equation:

Keep in mind that polarity notation is a convention and not a physical rule. As long as polarities are treated consistently, the correct answer will result. Figure 12.2 shows an example of a loop circuit wherein consistent polarity notation is critical to a correct answer. Two voltage sources are present in this circuit, but they are inserted with different polarities.

Circulating around the loop clockwise starting from the ground node, there is a 10-V rise, a voltage drop through the resistor, and a 5-V drop through the voltage source. The loop equation for this circuit is written as follows, yielding I_{LOOP} = 0.1 A:

Loop analysis in the context of a single loop circuit may not sound very different from the basics of Ohm’s law. It can be truly helpful when multiple loops are present in a circuit. The double-loop circuit in Fig. 12.3 is a somewhat contrived example but one that serves as a quick illustration of the concept. There are thrHfT—x9GOfthen88—x9G—Uon of

fully simplified, its overall current flow and power dissipation can be calculated: I = 54.5 mA and P = 545 mW.

Power dissipation for the resistors in Fig. 12.4 can be determined in multiple ways: VI, I^2R, or $V^2 \div R$. Power for each resistor can be calculated individually, but care must be taken to use the true voltage drop or current through each component. R1 and R2 have the same current passing through them because both have no parallel components to divert current, but they have differing voltage drops because their resistances are not equal. In contrast, R3 and R4 have identical voltage drops because they connect the same two nodes, but differing currents because their resistances are not equal.

The two resistor pairs, R1/R2 and R3/R4, form a basic voltage divider at the intermediate node connecting R2 and R3/R4. This voltage can be calculated knowing the current through R1 and R2 (54.5 mA) by either calculating the combined voltage drop of R1 and R2 and then subtracting this from the battery voltage or by just calculating the voltage drop across R3 and R4. The parallel combination of R3 and R4 equates to 33.3 Ω, indicating a voltage drop of 1.82 V at I = 54.5 mA. This is the voltage of the intermediate node because the lower node of R3 and R4 is ground, or 0 V. The alternate approach yields the same answer.

$$V_{NODE} = V_{BATT} - I\,(\mathrm{R1} + \mathrm{R2}) = 10\text{ V} - 54.5\text{ mA }(150\ \Omega) = 10\text{ V} - 8.18\text{ V} = 1.82\text{ V}$$

12.4 CAPACITORS

Resistors respond to changes in current in a linear fashion according to Ohm's law by exhibiting changes in voltage drop. Similarly, changing the voltage across a resistor causes the current through that resistor to change linearly. Resistors behave this way because they do not store energy; they dissipate some energy as heat and pass the remainder through the circuit. *Capacitors* store energy, and, consequently, their voltage/current relationship is nonlinear.

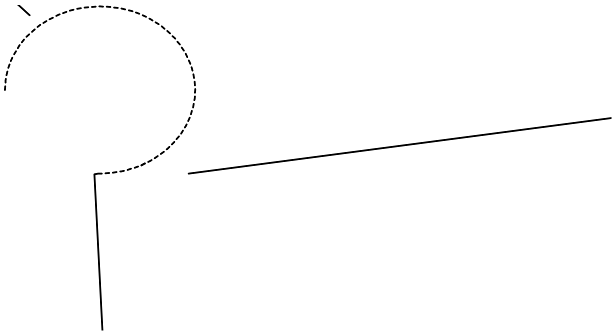
A capacitor stores charge on parallel conductive plates, each of which is at a different arbitrary potential relative to the other. In this respect, a capacitor functions like a very small battery and holds a voltage according to how much charge is stored on its plates. Capacitance (C) is measured in *farads*. One farad of capacitance is relatively large. Most capacitors that are used in digital systems are measured in microfarads (μF) or picofarads (pF). As a capacitor builds up charge, its voltage increases in a linear fashion as defined by the equation, $Q = CV$, where Q is the charge expressed in coulombs.

One of the basic demonstrations of a capacitor's operation is in the common series resistor-capaccreases, the voltage drop across the resistor decreases, causing the current through the circuit to decrease as well. Therefore, the capacitor begins charging at a high rate when its voltage is 0 and the circuit's current is limited only by the resistor. The charging rate decreases as the charge on the capacitor builds up.

Normalized to 1 V, the voltage across a capacitor in an RC circuit is defined as

where *e* is the base of the natural logarithm, an irrational mathematical constant roughly equivalent to 2.718. Starting from the initial condition when the capacitor is fully discharged, $t = 0$ and $V_C = 0$. The *RC time constant*, expressed in seconds, is simply the product of R and C and is a measure of how fast the capacitor charges. Every RC seconds, the voltage across the capacitor's terminals changes by 63.2 percent of the remaining voltage differential between the initial capacitor voltage and the applied voltage to the circuit, in this case the 10-V battery. By rule of thumb, a capacitor is often considered fully charged after 5 RC seconds, at which point it achieves more than 99 percent of its full charge. In this example, RC = 1,100 µs = 1.1 ms. Therefore, the capacitor would be at nearly 10 V after 5.5 ms of connecting the battery to the circuit.

RC circuits are used in timer applications where low cost is paramount. The accuracy of an RC timer is relatively poor, because capacitors exhibit significant capacitance variation, thereby altering the time constant. A simple oscillator can be constructed using an inverter (RC88Ate-=HDld8—A-=HDOHf



tors exhibit some series resistance and inductance, and all nearby pairs of conductors exhibit some mutual capacitance. A resistor consists of a resistive element encapsulated in some packaging material with a relatively small conductor at each end to connect the resistive element to an external circuit. Depending on the type of resistor, the conductors may be wires (leaded resistor) or small pieces of metal foil (surface mount). The resistive element itself will vary in size according to its power rating, material, and desired resistance. The finite lengths of the connecting leads and the resistive element each contribute a small quantity of inductance. There is also a small capacitance between the resistor's leads. These unwanted extras are called *parasitic* properties, because they usually detract from the performance of a system rather than improving it. A resistor's function is to provide a certain resistance value in a circuit, but its physical construction results in finite parasitic inductance and capacitance. Figure 12.11 shows a model of a nonideal resistor that enables analysis of its parasitic properties.

Each type of resistor exhibits different magnitudes of parasitic properties. Applications at lower frequencies often ignore these properties, because the parasitic inductance and capacitance is negligible as a result of the frequency/impedance relationships of inductors and capacitors. As the frequencies involved increase, series inductance is generally the first problem that is encountered. Inductance is minimized in resistors that have small leads or, better yet, no leads at all, as is the case with surface mount resistors. Inter-lead capacitance does not become a problem until frequencies get significantly higher.

Similarly, a capacitor exhibits parasitic resistance and inductance. The conductors that form the capacitor have finite resistance and inductance associated with them. A nonideal model of a capacitor is shown in Fig. 12.12. Inductance figures into a capacitor in much the same way that it does a resistor. Smaller leads and components result in reduced parasitic inductance.

At high frequencies, however, the capacitor's parasitic inductance has noticeable effects. The earlier example of using a capacitor to filter high-frequency noise showed that the capacitor removed most of the noise, but not all of it. As the frequency rises, the capacitor's impedance steadily decreases as expected. At a certain point, however, the frequency becomes high enough to cause noticeable impedance resulting from parasitic inductance. As the frequency continues to rise, the impedance of the capacitor begins to be more affected by the inductance. Because inductors resist high-frequency signals by increasing their impedance, the filtering capacitor loses its effect above a certain frequency limit. Figure 12.13 shows the general curve of impedance versus frequency for a capacitor. The curve shows that, above a certain frequency, a capacitor no longer behaves as expected from an ideal perspective. This threshold frequency is different for each type of capacitor and is determined by its physical construction. This is why power filter (e.g., decoupling or bypass) capacitors are ideally chosen based on the expected frequencies of noise that they are expected to attenuate.

As with parasitic inductance, a capacitor's parasitic resistance is a function of its physical construction. The industry-standard term that specifies this attribute is *equivalent series resistance* (ESR). Certain applications for capacitors tend to be more sensitive to ESR than others. ESR is generally not a major concern in high-frequency decoupling applications. However, when power-supply ripple needs to be attenuated, such as in a switching power supply, low-ESR capacitors may be critical to a successful circuit.

Inductors are subject to parasitic properties as well, in the form of series resistance in the wire coil and capacitance between individual coil windings and between the terminals. The nonideal inductor looks very much like the nonideal resistor in Fig. 12.11. Inductors that are used to filter power

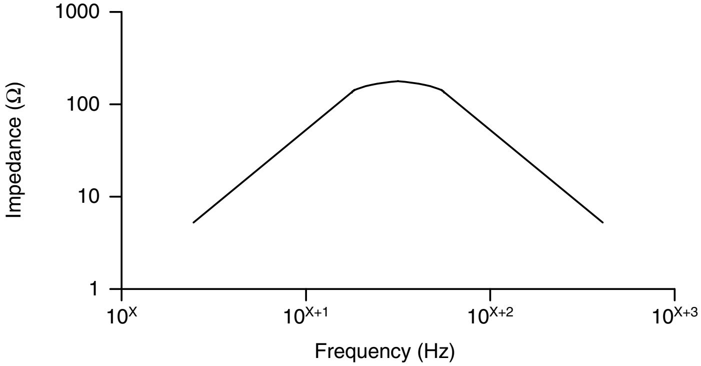


FIGURE 12.14 Nonideal inductor impedance vs. frequency curve.

desired in digital system applications that use inductors. Inductor manufacturers specify an inductor with a certain *self-resonant frequency* (SRF) to characterize the detrimental effects of the parasitic capacitance. Above the SRF, the inductor's impedance declines with increasing frequency. Therefore, if an inductor is operated near its SRF, its parasitic properties should be investigated to ensure that unexpected behavior does not result.

Many filtering applications in digital systems benefit from surface mount *ferrite* beads or chips. Ferrite is a magnetic ceramic material that behaves like an inductor: its impedance rises with frequency. A ferrite bead's parasitic capacitance is lower than that of a standard inductor, because there are no wire coils to capacitively couple with one another. Ferrites are suitable for attenuating high-frequency noise on power supplies and other signals, because they typically have high SRFs. A variety of ferrite materials exist with peak impedances at different frequencies to suit specific applications.

12.8 FREQUENCY DOMAIN ANALYSIS

Electrical signals on a wire can be viewed with an oscilloscope as a plot of voltage (or current) versus time. This is a *time-domain* view of the signals and it provides much useful information for a digital system designer. Using an oscilloscope, an engineer can verify the proper timing of a clock signal and its associated data and control signals. However, time-domain analysis is not very good at determining the frequency content of complex electrical signals. AC components are selected based on their impedances at certain frequencies. Therefore, a method is needed of evaluating a signal's frequency content, thereby knowing the frequencies of interest that the components must handle and enabling selection of suitable values.

Frequency-domain analysis enables an understanding of exactly how an overall AC circuit and its individual components respond to various frequencies that are presented to them. A frequency-domain view of a complex signal allows an engineer to tailor a circuit precisely to the application by relating frequencies and amplitudes rather than time and amplitudes in a conventional time-domain view. Pure sine waves are a convenient representation for signals, because they are easy to manipulate mathematically. While most real-world signals are not sine waves, Joseph Fourier, an eighteenth century French mathematician, demonstrated that an arbitrary signal (e.g., a microprocessor's square wave clock signal) can be expressed as the sum of many sine waves. Frequency-domain analysis is

Oscilloscopes are used to view signals in the time domain, whereas *spectrum analyzers* are used to view signals in the frequency domain. Figure 12.16 shows an example of a frequency-domain view of an electrical signal as observed on a spectrum analyzer (courtesy of Agilent Technologies). Rather than viewing voltage versus time, amplitude versus frequency is shown. Time is not shown, because the signals are assumed to be repetitive. Clearly, AC circuits operate on both repetitive and nonrepetitive signals. The analysis assumes repetitive signals, because an AC circuit's response is continuous. It does not have the ability to recognize sequences of signals in a digital sense and modify its behavior accordingly. Pure sine waves are represented by a vertical line on a frequency domain plot to indicate their amplitude at a single specific frequency. Since most real-world signals are not perfect sine waves, it is common to observe a frequency distribution around a single central frequency of interest.

While not strictly necessary, frequency-domain plots are usually drawn with *decibel* (dB) scales that are inherently logarithmic. The decibel is a relative unit of measurement that enables the comparison of power levels *(P)* entering and exiting a circuit. On its own, a decibel value does not indicate any absolute power level or measurement. A decibel is defined as a ratio of power entering and leaving a circuit:

$$\mathrm{dB} = 10\log_{10}\frac{P_{OUT}}{P_{IN}}$$

When the input and output power are identical, a level of 0 dB is achieved. Negative decibel levels indicate attenuation of power through the circuit, and positive decibel levels indicate amplification.

[redacted]

FIGURE 12.16 Spectrum analyzer frequency/amplitude plot. *(Reprinted with permission from Agilent Technologies.)*

dB at f_C. Because a passive filter has an ideal gain of zero at DC (no amplification), its gain at f_C is –3 dB.

However, the gain expression for this filter is in terms of voltage, not power. The relationship between voltage gain and decibel level is a multiplier of 20 rather than 10. Therefore, rather than looking for a gain of one-half at f_C, we must find a gain equal to the square root of one half because

$$-3 \text{ dB} = 20\log_{10}\frac{1}{\sqrt{2}}$$

The filter's gain expression is complex and must be converted into a real magnitude. Conveniently, the magnitude representation looks fairly close to what we are looking for.

$$|A_{FILTER}| = \frac{1}{\sqrt{1+(2\pi fRC)^2}}$$

By setting the denominator equal to the square root of 2, it can be observed that a gain of –3 dB is achieved when 2πf = 1 ÷ RC. Furthermore, it can be shown that the lowpass filter's gain declines at the rate of 20 dB per radian frequency decade because, for large *f*, the constant 1 in the denominator has an insignificant affect on the magnitude of the overall expression,

$$20\log_{10}\frac{1}{2\pi fRC}$$

Rather than trying to keep all of these numbers in one's head while attempting to solve a problem, it is common to plot the transfer function of a filter on a diagram called a *Bode plot*, named after the twentieth century American engineer H. W. Bode. A Bode magnitude plot for this lowpass filter is shown in Fig. 12.19. Filters and AC circuits in general also modify the phase of signals that pass through them. More complete AC analyses plot a circuit's phase transfer function using a similar Bode phase diagram. Phase diagrams are not shown, because these topics are outside the scope of this discussion.

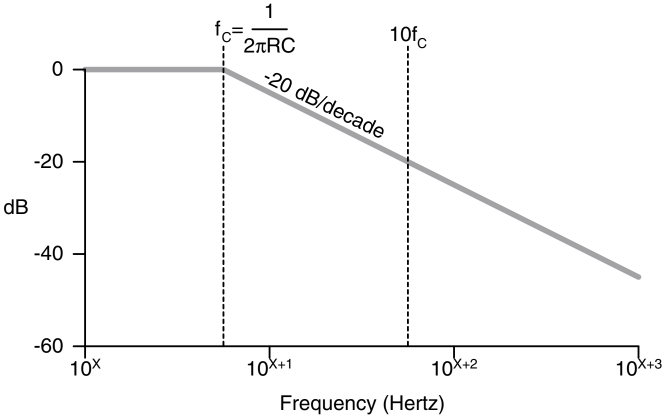


FIGURE 12.19 Bode magnitude plot for RC lowpass filter.

This Bode plot is idealized, because it shows a constant gain up to f_C and then an abrupt roll-off in filter gain. A real magnitude calculation would show a smooth curve that conforms roughly, but not exactly, to the idealized plot. For first-order evaluations of a filter's frequency response, this idealized form is usually adequate.

A filter's *passband* is the range of frequencies that are passed by a filter with little or no attenuation. Conversely, the *stopband* is the range of frequencies that are attenuated. The trick in fitting a filter to a particular application is in designing one that has a sufficiently sharp roll-off so that unwanted frequencies are attenuated to the required levels, and the desired frequencies are passed with little attenuation. If the passband and desired stopband are sufficiently far apart, a simple first-order filter will suffice, as shown in Fig. 12.20. Here, the undesired noise is almost three decades beyond the signal of interest. The RC filter's 20-dB/decade roll-off provides an attenuation of up to 60 dB, or 99.9 percent, at these frequencies.

As the passbands and stopbands get closer together, more complex filters with sharper roll-offs are necessary to sufficiently attenuate the undesired frequencies while not disturbing those of interest. Incorporating additional AC elements into the filter design can increase the slope of the gain curve beyond f_C. A second-order lowpass filter can be constructed by substituting an inductor in place of the series resistor in a standard RC circuit as shown in Fig. 12.21.

The LC filter's gain can be calculated as follows:

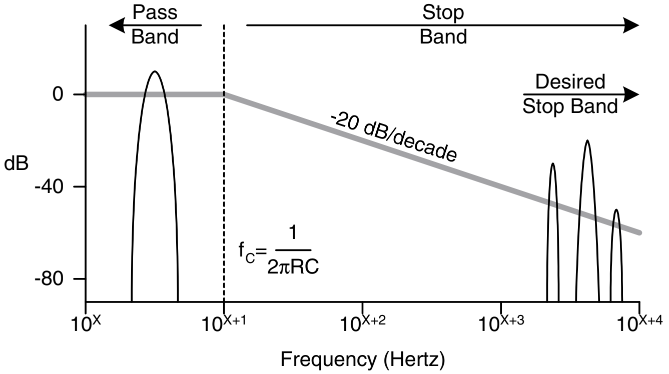
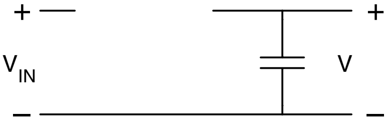


FIGURE 12.20 Widely separated pass and stopbands.



The cutoff frequency is determined as was done previously for the first-order filter, for which the magnitude of the transfer function's denominator is the square root of 2. To meet this criterion, 2πf = $1/\sqrt{LC}$. The additional AC element in the filter introduces a frequency-squared term that doubles the slope of the gain curve beyond f_C. Therefore, this second-order lowpass filter declines at 40 dB per decade instead of just 20 dB per decade.

Higher-order filters can be created by adding LC segments to the basic second-order circuit to achieve steeper gain curves as shown in Fig. 12.22. These basic topologies are commonly referred to as *T* and *pi* due to their graphical resemblance to the two characters.

Lowpass filters are probably the most common class of filters used in purely digital systems for purposes of noise reduction. However, when analog circuits are mixed in, typically for interface applications including audio and radio frequencies, other types of filters become useful. The inverse of a lowpass filter is a *highpass* filter, which attenuates lower frequencies and passes higher frequencies. A first-order RC highpass filter is very similar to the lowpass version except that the topology is reversed as shown in Fig. 12.23. Here, the capacitor blocks lower frequency signals but allows higher frequencies to pass as its impedance drops with increasing frequency.

As done previously, the transfer function can be calculated by combining the impedances of each element into a single expression:

$$A = \frac{Z_R}{Z_R + Z_C} = \frac{R}{R + \frac{1}{2\pi fC}} = \frac{2\pi fRC}{2\pi fRC + 1}$$

It can be observed from the transfer function that, as the frequency approaches DC, the filter's gain approaches 0. At higher frequencies, the filter's gain approaches 1, or 0 dB. First- and second-order highpass filters have gain slopes of 20 and 40 dB per decade, respectively, as mirror images of the lowpass filter frequency response. A second-order highpass filter can be created by substituting an appropriate inductor in place of the shunt resistor. The inductor appears as a short circuit to the negative voltage rail at low frequencies and gradually increases its impedance as the frequency rises. At high frequencies, the inductor's impedance is sufficiently high that it is shunting almost no current to the negative voltage rail.

Just as for lowpass filters, higher-order highpass filters can be created by adding LC segments in T and pi topologies, albeit with the locations of the inductors and capacitors swapped to achieve the highpass transfer function.

12.10 BANDPASS AND BAND-REJECT FILTERS

Some signal manipulation applications require the passing or rejection of a selective range of frequencies that do not begin at DC (a lowpass filter) nor end at an upper limit that is conceptually in-

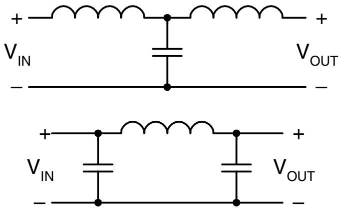


FIGURE 12.22 Third-order LC lowpass filters.

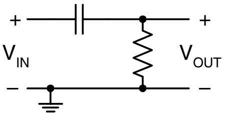


FIGURE 12.23 First-order RC highpass filter.

finity (a highpass filter). When a radio or television is tuned to a certain channel, a *bandpass* filter selects a certain narrow range of frequencies to pass while attenuating frequencies above and below that range. Bandpass filters may have limited utility in a typical digital system, but certain interface circuitry may require bandpass filtering to attenuate low-frequency and DC content while also reducing high frequency noise.

Bandpass filter design can get fairly complex, depending on the required AC specifications. Design issues include the width of the passband and the slope of the gain curve on either side of the passband. Two basic bandpass topologies are shown in Fig. 12.24. These are second-order circuits that pair an inductor and capacitor together in either a series or shunt configuration. The series topology operates by blocking the high-frequency stopband with the inductor and the low-frequency stopband with the capacitor. In the middle is a range of frequencies that are passed by both elements. The shunt topology operates by diverting the high-frequency stopband to ground with the capacitor and the low-frequency stopband with the inductor. This topology takes special advantage of the parallel LC resonant circuit mentioned previously.

Each of these topologies can be designed with passbands of arbitrary width. The bandwidth of the filter has a direct impact on the shape of its transfer function. However, for a second-order bandpass filter, each AC element contributes a –20 dB per decade roll-off on either side of the center frequency. Therefore, a narrow ideal second-order bandpass filter has the transfer function shown in Fig. 12.25, and its center frequency is

$$f_C = \frac{1}{2\pi\sqrt{LC}}$$

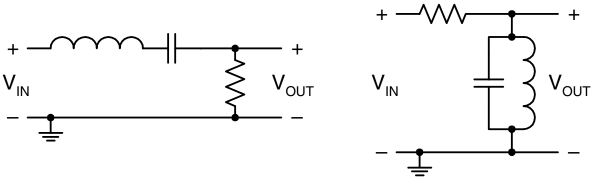


FIGURE 12.24 Second-order bandpass filter topologies.

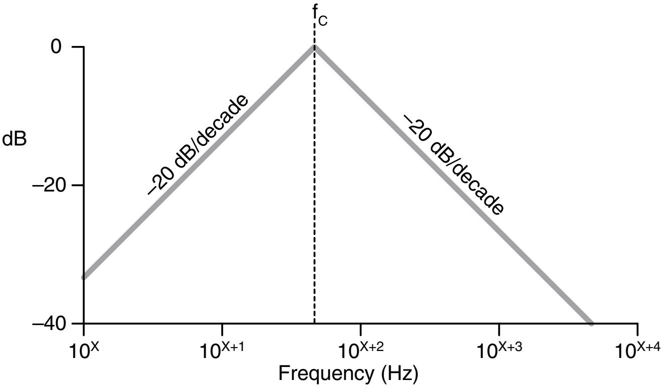


FIGURE 12.25 Idealized bandpass filter frequency response.

$$\frac{V_{PRIMARY}}{V_{SECONDARY}} = \frac{N_{PRIMARY}}{N_{SECONDARY}}$$

A transformer is a passive component; it has no capability of amplifying a signal. Consequently, an ideal transformer passes 100 percent of the power applied to it and satisfies the equation $V_S I_S = V_P I_P$. If the primary coil has more windings than the secondary coil, an AC signal of lesser magnitude will be induced on the secondary coil when an AC signal is applied to the primary coil. The current flowing through the secondary coil will be higher than that in the primary so that conservation of energy is preserved. In reality, of course, a transformer has less than 100 percent efficiency as a result of parasitic properties, including finite resistance of the coils and less-than-perfect magnetic coupling.

One of the most common uses of a transformer is in power distribution in which AC power is either stepped up or stepped down, depending on the application. Figure 12.27 shows a basic transformer with a 120 VAC signal injected into the primary coil and a load resistor on the secondary coil. The ratio of the primary to secondary windings is 10:1; perhaps the primary coil has 1,000 windings whereas the secondary has 100. The result is a step-down of high-voltage power from a wall outlet to a more manageable 12 VAC. This illustrates why AC power distribution is so convenient: voltages can be arbitrarily transformed without any complicated electronic circuits. It is advantageous to distribute power at a higher voltage to reduce the current draw for a given power level. Lower current means lower I^2R power losses in the distribution wiring. The 12 VAC transformer output may power the voltage regulator of a digital circuit. If the digital circuit draws 10 A at 12 VAC, it will draw only 1 A at 120 VAC.

Transformers are critical to power distribution both at the system level and at the generation and utility levels. Power is stepped up at generating plants with transformers to as high as 765,000 V for efficient long-distance distribution. As the power gets closer to your home or office, it is stepped down to intermediate levels and finally enters the premises at 120 and 240 VAC.

Aside from power supply applications, transformers are also used for filtering and impedance matching of interface signals. The use of a transformer as a filter requires knowledge of the physical orientations of the primary and secondary windings with respect to one another. Each winding in a transformer has two terminals. When a signal is applied to the primary coil, a decision is made as to which terminal is connected to the positive portion of the circuit and which is connected to the negative portion (ground in some applications). That signal will induce a signal of equivalent polarity on the secondary coil when the appropriate choice of positive and negative terminals is made at the other end of the transformer. Alternatively, a signal of opposite polarity will be induced if the secondary coil's positive and negative connections are swapped. The graphical convention of distinguishing the matching terminals of the primary and secondary coils is by placing matching reference dots next to one terminal of each coil as shown in Fig. 12.28. The coils may be drawn with their dots on the same side or on opposite ends of the transformer.

The relative polarity of the primary and secondary windings is important for filtering applications to ensure that magnetic fields in each coil either add to or cancel each other as appropriate. Trans-

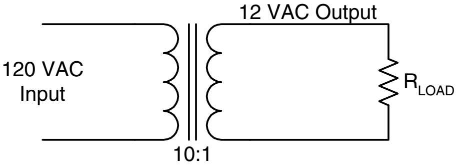


FIGURE 12.27 Basic transformer operation.

This page intentionally left blank.

CHAPTER 13
Diodes and Transistors

Most of the semiconductors that a digital system employs are fabricated as part of integrated circuits. Yet there are numerous instances in which discrete semiconductors, most notably diodes and transistors, are required to complete a system. Diodes are found in power supplies, where they serve as rectifiers and voltage references. It is difficult these days *not* to find a light emitting diode, or LED, in one's immediate vicinity, on some appliance or piece of electronic equipment. Discrete transistors are present in switching power supplies and in circuits wherein a digital IC must drive a heavy load. There are many other uses for diodes and transistors in analog circuit design, most notably in signal amplification. These more analog topics are not discussed here.

Diodes and transistors are explained in this chapter from the perspective of how they are applied in the majority of digital systems. As such, the level of theory and mathematics used to explain their operation is limited. The first portion of the chapter introduces diodes and provides examples of how they are used in common power and digital applications. Bipolar junction and field-effect transistors are discussed in the remainder of the chapter. The intent of this chapter is to show how diodes and transistors can be put to immediate and practical use in a digital systems context. As such, useful example circuits are presented whenever possible.

13.1 DIODES

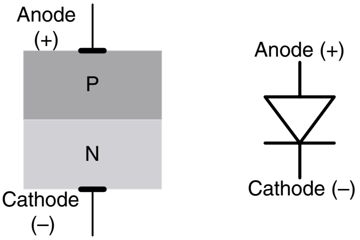


FIGURE 13.1 Diode structure and graphical representation.

An ideal diode is a nonlinear circuit element that conducts current only when the device is *forward biased*, i.e., when the voltage applied across its terminals is positive. It thereby behaves as a one-way electrical valve that prevents current from flowing under conditions of *reverse bias*. A diode has two terminals: the *anode* and *cathode*. For the diode to be forward biased, the anode must be at a more positive voltage than the cathode. Diodes are the most basic semiconductor structures and are formed by the junctions of two semiconductor materials of slightly differing properties. In the case of a silicon diode, the anode is formed from positively doped silicon, and the cathode is formed from negatively doped silicon. Along this *pn junction* is where the physical phenomenon occurs that creates a diode. Figure 13.1 shows the general silicon structure of a diode and its associated symbolic representation.

Real diodes differ from the ideal concept in several ways. Most significantly, a real diode must be forward biased beyond a certain threshold before the device will conduct. This threshold is called the

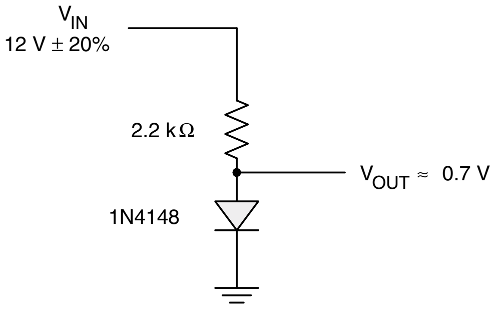
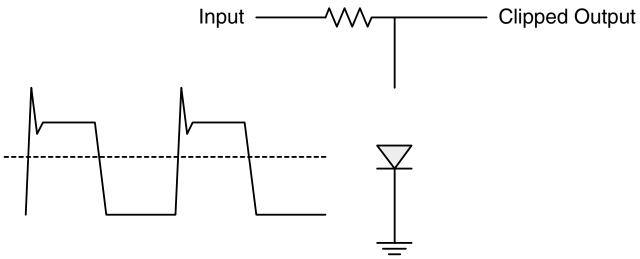


FIGURE 13.3 Diode-based voltage reference.

5 mA under typical conditions.[*] The 2.2-kΩ resistor limits the current through the diode to approximately 5 mA when V_{IN} = 12 V. If the input changes by 20 percent and causes a corresponding change in the current, the diode voltage changes by a small fraction. Using a basic small signal diode in this manner is an effective scheme for many applications. If tighter tolerance is desired, more stable voltage reference diodes are available. Thermal overload is not a problem for this diode, because its power dissipation is relatively constant at 0.7 V × 5 mA = 3.5 mW.

Diodes are available with a broad spectrum of characteristics. Aside from silicon diodes, there are *Schottky* diodes that exhibit lower forward voltages of under 0.5 V. Lower forward voltages provide benefits for high-power applications in which heat and power dissipation are prime concerns. Reduced V_F means reduced power. Diodes are manufactured in a variety of packages according to the amount of power that they are designed to handle. Small-signal diodes are not intended to handle much power and are available in small, surface mount packages. At the other extreme, diodes can be as large as hockey pucks for very high-power applications. Small-signal diodes are manufactured with varying response times to changes in voltage. A diode can be used to *clip* a signal to prevent it from exceeding a certain absolute voltage, as shown in Fig. 13.4. As the signal's edge rate increases, a slower diode may not respond fast enough to be effective. If a single diode's forward voltage is insufficient, multiple diodes can be placed in series to increase the clipping threshold. Some of the more common small-signal diodes used in digital circuits include the leaded 1N914 and 1N4148 devices, and their surface mount equivalents, the SOT-23 MMBD914 and MMBD4148.



[*] 1N4148, Fairchild Semiconductor Corporation, 2002, p. 2.

13.2 POWER CIRCUITS WITH DIODES

A major use for diodes is in the *rectification* of AC signals, specifically in power supplies in which the conversion from AC to DC is required. Small-signal diodes can be used as rectifiers in non-power or low-power applications. Larger diodes with higher power ratings are employed when constructing power supply circuits meant to provide more power. An AC power signal is a sine wave of arbitrary amplitude that is centered about 0 V. Its voltage peaks are of equal magnitude above and below 0 V. A digital circuit requires a steady DC power supply. The first step in creating a steady DC power supply is to rectify the AC input such that the negative AC sine wave excursions are blocked. Figure 13.5 shows a single diode performing this function. The rectified output is reduced in voltage by the diode's forward voltage. This circuit is called a *half-wave* rectifier, because it passes only half of the incoming power signal. Once rectified, capacitors and inductors can smooth out (lowpass filter) the rectified AC signal to create a steady DC output.

The single-diode half-wave rectifier does the job, but does not take advantage of the negative portion of the AC input. Four diodes can be assembled into a *full-wave bridge rectifier* that passes the positive portion of the sine wave and inverts the negative portion relative to the DC ground. This circuit is shown in Fig. 13.6. The bridge rectifier works by providing a current conduction path through the resistor to ground regardless of the polarity of the incoming AC signal. When the AC input is positive with respect to the polarity markings shown in the diagram, diodes D1 and D3 are forward biased, conducting current from D1 through the resistor, then through D3 to the negative AC input wire. When the AC input is negative during the next half of the sinusoid, D2 and D4 are forward biased and allow current to flow in the same direction through the resistor. The result is that a positive voltage is always developed across the load with respect to ground. Note that, because of the two diodes in series with the load, the rectified output voltage is reduced by twice the diodes' forward voltage.

Power rectifier circuits are generally found in systems wherein a high-voltage input (e.g., 120 VAC) must be converted into a low-voltage output such as +5 VDC to power a digital logic circuit. Transformers are used in conjunction with bridge rectifiers to step down the high-voltage AC input to a more appropriate intermediate level that is much closer to the final voltage level required by the system. A power filter circuit can then be used to s©Hfw©HstD—x9AwH©flexH-…unoA thenGm…8

intermediate voltage into a more accurate digital supply voltage. This common AC-to-DC power supply configuration is illustrated in Fig. 13.7.

Another power application of diodes is in combining multiple power supplies to feed a single component or group of components while ensuring that the failure or disappearance of one supply does not cause that component to lose power and cease operation. This concept relies on the fact that a standard diode will not conduct under normal reverse-bias conditions. As shown in Fig. 13.8, each power supply is isolated by a diode whose cathodes form a common voltage supply node for a circuit. Under normal operating conditions, each diode is forward biased, because the respective power supplies are functioning. When one supply fails, its associated diode becomes reverse biased, thereby preventing the failing supply from pulling power from the functioning supply and causing the system to fail. These diodes are often called *OR-ing diodes,* because they perform a logical OR function on the power supplies.

Diode OR-ing circuits are also seen in battery-backup applications in which it is desired to keep a low-power static RAM chip powered by a battery when the main power supply is turned off. A typical scenario is a higher-voltage operating supply (e.g., +5 V) and a lower-voltage data-retention bat-

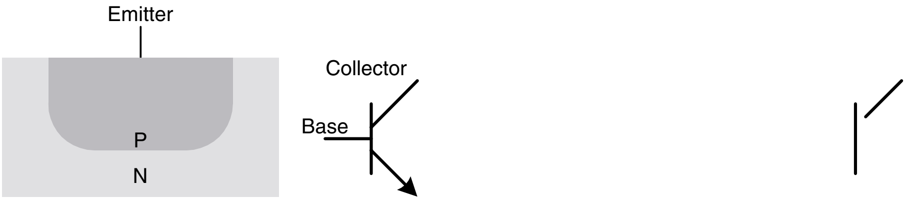
13.4 BIPOLAR JUNCTION TRANSISTORS

Transistors are silicon switches that enable a weak signal to control a much larger current flow, which is the process of amplification: magnifying the amplitude of a signal. *Bipolar junction transistors* (BJTs) are a basic type of transistor and are formed by two back-to-back pn junctions. Figure 13.11 shows the general BJT structures and their associated symbolic representations. The BJT consists of three layers, or regions, of silicon in either of two configurations: NPN and PNP. The middle region is called the *base*, and the two outer regions are separately referred to as the *collector* and *emitter.* As will soon be shown, the base-emitter junction is what enables control of a potentially large current flow between the collector and emitter with a very small base-emitter current. A BJT's construction is more than simply placing two diodes back to back. The base region is extremely thin to enable conduction between the collector and emitter, and the collector and emitter are sized differently according to the fabrication process.

Currents in an NPN transistor flow from the base to the emitter and from the collector to the emitter. The relationship between these currents is defined by a proportionality constant called beta (β, also known as h_{FE}): $I_C = \beta I_B$. Beta is specific to each type of transistor and is characterized by the manufacturer in data sheets. Typical values for beta are from 100 to less than 1000. The beta current relationship provides a quick view of how a small base current can control a much larger collector current. A higher beta indicates greater potential for signal amplification. Because the base-emitter junction is essentially a diode, it must be sufficiently forward biased for the transistor to conduct current (V_{BE} = 0.7 V under typical conditions). A PNP transistor functions similarly, although the polarities of currents and voltages are reversed.

When a transistor circuit is designed, care must be taken not to overdrive the base-emitter junction. Like any other diode, it presents very low impedance beyond its forward voltage. Without some type of current limiting, the transistor will overheat and become damaged. Transistors are biased using resistors placed at two or three of its terminals to establish suitable operating voltages. Figure 13.12 shows a common NPN configuration at DC with a current limiting resistor, R_B, at the base and a voltage-dropping resistor, R_C, at the collector. The emitter is grounded, establishing the base voltage, V_B, at 0.7 V. R_B sets the current flowing into the base and thereby controls the collector voltage, V_C. As R_B increases, V_C increases, because less current is pulled through the collector, reducing the voltage drop across R_C. In this example, the base current, I_B, is (5 V – V_B) + R_B = 0.43 mA. Assuming a beta of 100, the collector current, I_C, is 43 mA, and V_C = 5 V – $I_C R_C$ = 2.85 V.

The transistor is limited in how much current it can drive by both its physical characteristics and the manner in which it is biased. Physically speaking, a transistor will have a specified maximum power dissipation beyond which it will overheat and eventually become damaged. In this circuit, the transistor's power dissipation is $V_{CE}I_C + V_{BE}I_B$, although the dominant term is between the collector and emitter, where the great majority of the current flow exists. Using this small simplification, the transistor is dissipating approximately 2.85 V × 43 mA = 123 mW.



logy.

ꞩtor is not operated beyond its physical limitations, the bias configuration ıow much current flows into the collector. A BJT has three modes of opera- *ɑturation*. In cutoff, the transistor is not conducting, because the base-emit- ɜrse biased or insufficiently forward biased. The collector is at its maximum ɔllector junction is reverse biased, because no current is flowing to create a ʔ$_C$ or its equivalent. When the base-emitter junction is forward biased, the ːrent and V_C begins to drop. The transistor is in active mode. As long as the ɑ remains reverse biased, increasing base current will cause a corresponding ːurrent, and the transistor remains in active mode. If I_B is increased to the point llector junction is forward biased (increased I_C causes V_C to approach V_E), the ɹration and no longer can draw more current through the collector. Saturation ɘ transistor, but it results in a nonlinear relationship between I_B and I_C, nullifying If R_C is increased or decreased, saturation occurs at lower or higher I_C, respec- ircuits must avoid saturation to function properly because of the resulting nonlin- d in a purely digital context, however, transistors can be driven from cutoff to ꞡ as the power dissipation specifications are obeyed.

CATION WITH THE BJT

; knowledge of BJT operation, an NPN transistor can be already be applied in a useful ıcation: driving a high-current LED array with a relatively weak output pin from a logic .l digital output pins have relatively low current drive capabilities, because they are de- ımarily to interface with other logic gates that have low input current requirements. CMOS ; tend to exhibit symmetrical drive currents in both logic 1 and logic 0. A CMOS output may ı for anywhere from several milliamps to tens of milliamps. Bipolar logic ICs tend to exhibit ɜly low drive current at logic 1, often less than one milliamp, and higher currents of several ımps at logic 0. The use of bipolar logic is widespread, and it is often advantageous to take ad- ɑge of the greater drive capability of the logic-0 state. Figure 13.13 shows a logic output con- ted directly to an LED with each of the two possible polarities. The active-low configuration ɾns the LED on when the output is logic 0, and the active-high turns the LED on when the output is ɔgic 1.

If neither the logic-1 nor logic-0 current capabilities are sufficient for the load, a simple transistor circuit can solve the problem. A typical 74LS logic family output pin is specified to source 0.4 mA at 2.7 V when driving a logic 1. Assuming a minimum beta of 100, an NPN transistor can be used to

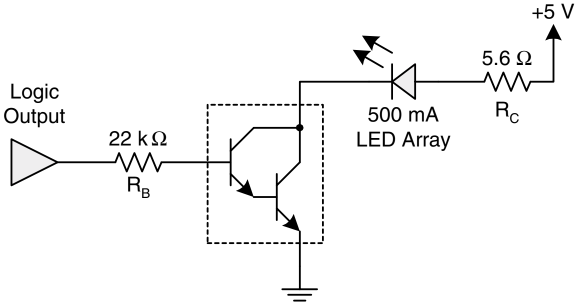


FIGURE 13.16 NPN Darlington pair LED driver.

transistor to the base of the second. As before, the betas multiply, but now V_{BE} is doubled to 1.4 V, because the two base-emitter junctions are in series. The higher V_{BE} reduces the base current from the previous example to roughly 0.06 mA, because the same value of R_B, 22 kΩ, is used. However, with an overall beta of approximately 10,000, the LED array is adequately driven.

13.6 LOGIC FUNCTIONS WITH THE BJT

The preceding circuits are binary amplifiers. In response to a small binary current input, a larger binary output current is generated. Looking at these circuits another way reveals that they are very basic logic gates: inverters. Consider the now familiar circuit in Fig. 13.17a. Rather than driving a high-current load, the output voltage is taken at the collector. When a TTL logic 0 is driven in, R_C pulls the output up to logic 1, 5 V. When a TTL logic 1 is driven in, the transistor saturates and drives the output down to logic 0, $V_{CE\,(SAT)}$. Logic gates in bipolar ICs are more complex than this, but the basic idea is that a simple inverter can be constructed with discrete transistors if the need arises. This discrete inverter can be transformed into a NOR gate by adding a second transistor in parallel with

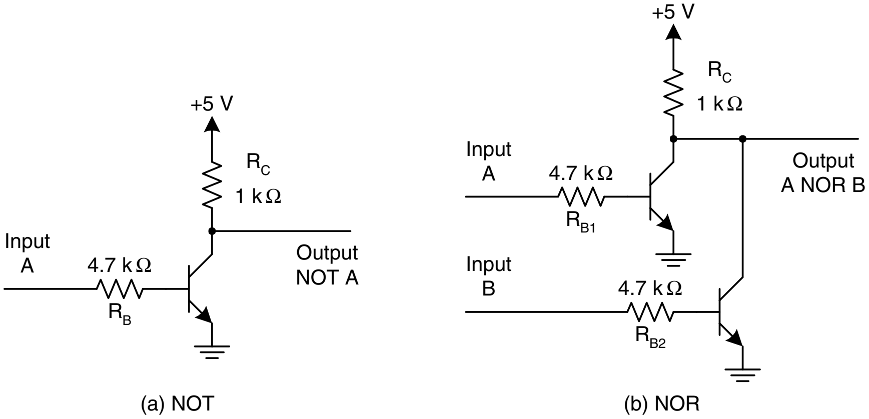


FIGURE 13.17 NPN NOT and NOR gates.

the one already present. In doing so, the output is logic 1 whenever both inputs are logic 0. As soon as one input is driven to logic 1, the accompanying transistor pulls the common V_C output node toward ground and logic 0. An advantage of creating logic gates from discrete transistors is that incompatible voltage domains can be safely bridged. In this example, the logic output is 5 V compatible, while the input can be almost any range of voltages as long as the transistor's specifications are not violated. This voltage conversion function would not be possible over such a wide range with normal bipolar or CMOS logic ICs.

It has been previously mentioned that TTL, or bipolar, logic outputs are asymmetrical in their 0 and 1 logic level drive strengths. A TTL output, shown in Fig. 13.18, consists of a *totem pole* output stage and a driver, or buffer, stage that passes the 0/1 logic function result to the output stage. It is called a totem pole output stage, because the vertical stack of two transistors and a diode somewhat resembles the layering of carvings on a totem pole. Classic TTL logic is composed entirely of NPN transistors, because PNP transistors are more difficult to fabricate on an integrated circuit.

When Q1 is turned on, its emitter voltage, V_{E1}, is fixed at 0.7 V by Q2. Q1 is driven into saturation by the logic circuit (not shown), which brings its collector voltage, V_{C1}, down to $V_{CE\ (SAT)} + V_{E1}$, which is approximately equal to 0.9 V. The saturation voltage of Q1 is approximately 0.2 V, because

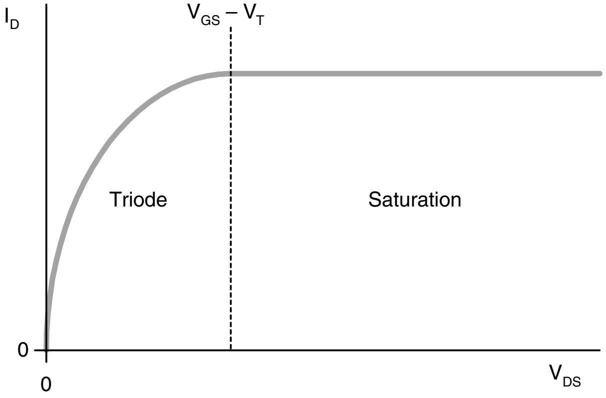


FIGURE 13.20 Enhancement-type n-FET I_D vs. V_{DS} for fixed $V_{GS} > V_T$.

The I_D/V_{DS} curve can be mathematically calculated, but the formulas require knowledge of specific physical parameters of a transistor's fabrication process. When integrated circuits are designed, such information is critical to device operation, and manufacturing process parameters are at an engineer's disposal. Data sheets for discrete FETs, however, do not typically provide the detailed process parameters required for these calculations. Instead, manufacturers provide device characterization curves in their data sheets that show I_D/V_{DS} curves for varying V_{GS}. An example of this is the graph contained in Fairchild Semiconductor's 2N7002 NMOS transistor data sheet and shown in Fig. 13.21. PMOS transistors function in the same manner as NMOS, although the polarities are reversed. The source is typically at a higher voltage than the drain, and V_{GS} is expressed as a negative value.

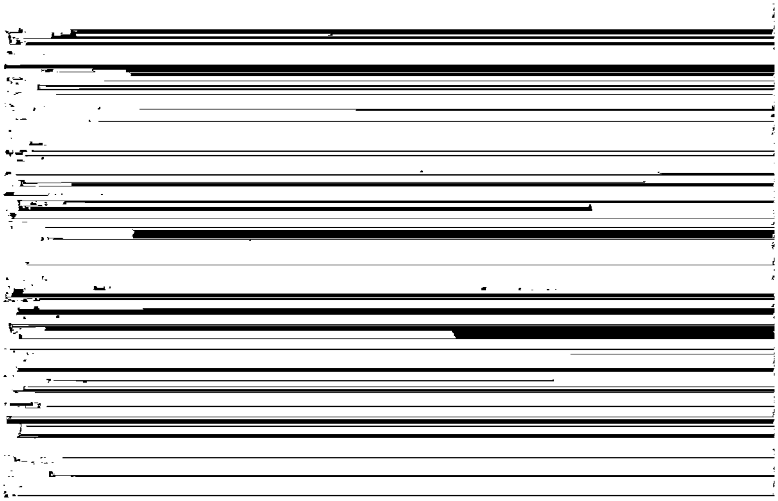


FIGURE 13.21 2N7002 I_D/V_{DS} graph. *(Reprinted with permission from Fairchild Semiconductor and National Semiconductor.)*

13.8 POWER FETS AND JFETS

Discrete FETs are used in a variety of applications. In digital systems, FETs are often found in power supply and regulation circuitry because of the availability of low-resistance devices. A key parameter of a FET used in power applications is its channel resistance between the source and drain, R_{DS}. Per the basic power relationship, P = I^2R, a FET with low R_{DS} will waste less power and will therefore operate at a cooler temperature. Power FET circuits can either handle more current without overheating or can be made to run cooler to extend their operational life span. It is not difficult to find power FETs with R_{DS} well below 10 mΩ. In contrast, a BJT exhibits a $V_{CE\ (SAT)}$ that dissipates significant power at high currents (P = IV). The saturation voltage also increases with I_C, causing more power to be dissipated in high-power applications.

When FETs are constructed as part of an IC, they are built in a lateral configuration atop the silicon substrate with a structure similar to what was shown earlier. Discrete FETs, however, are often constructed in a vertical manner known as *double-diffused MOS*, or DMOS, where the source and drain are on opposite sides of the silicon chip. The DMOS structure is shown in Fig. 13.24. DMOS surrounds the source with a thin region of oppositely doped silicon that serves as the body through which the conducting channel is induced by a voltage applied to the gate. Around the body is the substrate, which is doped similarly to the drain. Discrete FETs are constructed in this manner, because the thin channel between the source and the substrate provides low R_{DS} and, consequently, high current capacity with reduced power dissipation.

DMOS FETs are constructed in a manner that electrically connects the source and body regions, as shown by the metal source contacts in Fig. 13.24, so that a parasitic NPN transistor does not arise and cause problems. A consequence of this technique is that a parasitic diode is formed between the body and drain. Because the body is connected to the source, this is actually a source-drain diode with the anode at the source and the cathode at the drain for an n-FET. The diode is reverse biased under most conditions, because an n-FET's source is usually at a lower voltage than the drain. Therefore, V_{DS} would have to approach –0.6 V for conduction to occur. Figure 13.24 also shows the graphical representation of an n-type DMOS transistor with a source-drain diode. If an n-FET is designed such that the source is always connected to ground and the drain can never drop below 0 V, the diode has no effect. In less obvious configurations, the biasing of this inherent diode should be taken into account to ensure that current does not flow through an unintended path.

A potentially dangerous characteristic of FETs is a consequence of the insulated gate's high input impedance. If a FET circuit does not create a path to the gate through which stray electrical charge can drain away, a high voltage can build up and cause permanent damage to the device. The gate is a small capacitor, and it is known that V_{CAP} = Q ÷ C. Therefore, a small charge accumulating on a very

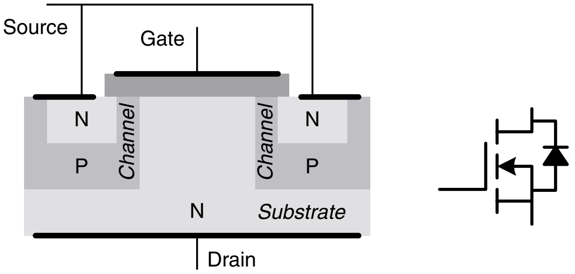


FIGURE 13.24 DMOS n-FET simplified structure and graphical representation.

small capacitor results in a high voltage. If the voltage exceeds the breakdown voltage of the gate's insulating silicon dioxide, it will break through and cause permanent damage to the FET. This is why MOS devices are particularly sensitive to static electricity. MOS components should be stored in conductive material that prevents the accumulation of charge on FET gate terminals.

Thus far, we have covered only enhancement-type MOSFETs. *Depletion-type* MOSFETs are built with a similar structure, but the channel region is doped with n-type silicon (in the case of an n-FET) so that the transistor conducts when $V_{GS} = 0$. Instead of defaulting to an open circuit, it conducts instead through the physically implanted channel. If positive V_{GS} is applied to a depletion-type n-FET, the channel is enhanced, and it can conduct more current as V_{DS} increases. However, if V_{GS} is made negative, the channel is depleted, and less current is conducted for a given V_{DS}. Schematic symbols for depletion-type MOSFETs are shown in Fig. 13.25. These devices are used in integrated circuits and are less common in discrete form.

Another type of FET, the *junction FET* (JFET) is not a MOS device and bears some similarity in structure to a BJT. As shown in Fig. 13.26, a JFET does not contain an insulated gate and does contain a physically implanted channel. Like a depletion-type MOSFET, a JFET conducts when $V_{GS} =$ 0, and decreasing V_{GS} depletes the channel. As a result of the lack of gate insulation, the gate-drain and gate-source junctions will conduct when forward biased, thereby negating the transistor's operation. In the context of integrated circuits, JFETs are used mainly in bipolar analog processes, because they provide a higher input resistance as compared to BJTs.

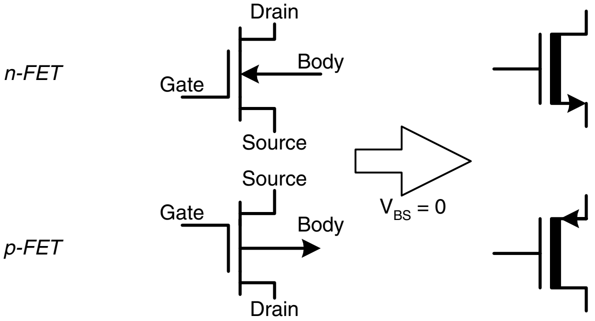


FIGURE 13.25 Depletion-type MOSFET graphical representations.

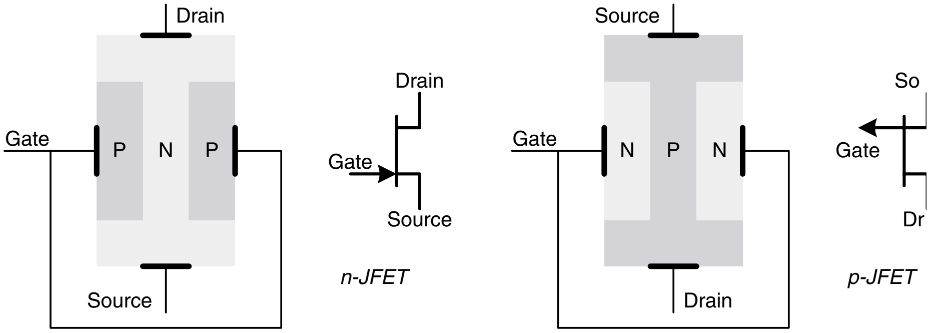


FIGURE 13.26 JFET structure and graphical representation.

CHAPTER 14
Operational Amplifiers

Transistors are the basic building blocks of solid-state amplifiers. Designing an amplifier using discrete transistors can be a substantial undertaking that requires theory outside the scope of this book. Operational amplifiers, or *op-amps,* exist to make the design of basic amplification circuits relatively easy. A digital engineer can use an op-amp to construct a general-purpose amplifier or active filter to preprocess an analog signal that may serve as input to the system. Comparators, which are based on op-amps, are useful in triggering events based on a signal reaching a certain threshold.

A key benefit of the op-amp is that it implements complex discrete transistor circuitry within a single integrated circuit and presents the engineer with a straightforward three-terminal amplifier that has well defined specifications and that can be externally configured to exhibit a wide range of characteristics.

Op-amps are presented here from three basic perspectives. First, the device is introduced using an idealized model so that its basic operation can be explained clearly without involving too many simultaneous details. The ideal op-amp is a very useful construct, because many real op-amp circuits can be treated as being ideal, as will be demonstrated later. Fundamental op-amp circuit analysis is stepped through in detail as part of the ideal presentation. The second section brings in nonideal device behavior and discusses how the idealized assumptions already introduced are affected in real circuits. The remainder of the chapter walks through a broad array of common op-amp circuit topologies with step-by-step analyses of their operation. The last of these presentations deals with the op-amp's cousin, the comparator, and explains the important topic of hysteresis.

14.1 THE IDEAL OP-AMP

The design of amplifiers is normally most relevant in analog circuits such as those found in audio and RF communications. An amplifier is an analog circuit that outputs a signal with greater amplitude than what is presented to it at the input. Amplification is sometimes necessary in digital systems. Amplifiers are often found at interfaces where the weak signal from a transducer (e.g., microphone or antenna) must be strengthened for sampling by an analog-to-digital converter. Even if a signal has sufficient amplitude, it may be desirable to scale it for better sampling resolution. For example, if an analog-to-digital converter accepts an input of 0 to 5 V and the incoming signal swings only between 0 and 3 V, 40 percent of the converter's resolution will be wasted. An amplifier can be used to scale the signal up to the full 5-V input range of the converter.

Solid-state amplifiers are constructed using transistors integrated onto a silicon chip or discrete transistors wired together on a circuit board. Amplifiers range greatly in complexity; complete AC analysis theory and its application to discrete amplifier design are outside the scope of this book. However, the design of many general-purpose amplifiers is made easier by the availability of prebuilt components called *operational amplifiers* (op-amps). Op-amps are so common that they are

pensate for, V_{BIAS}

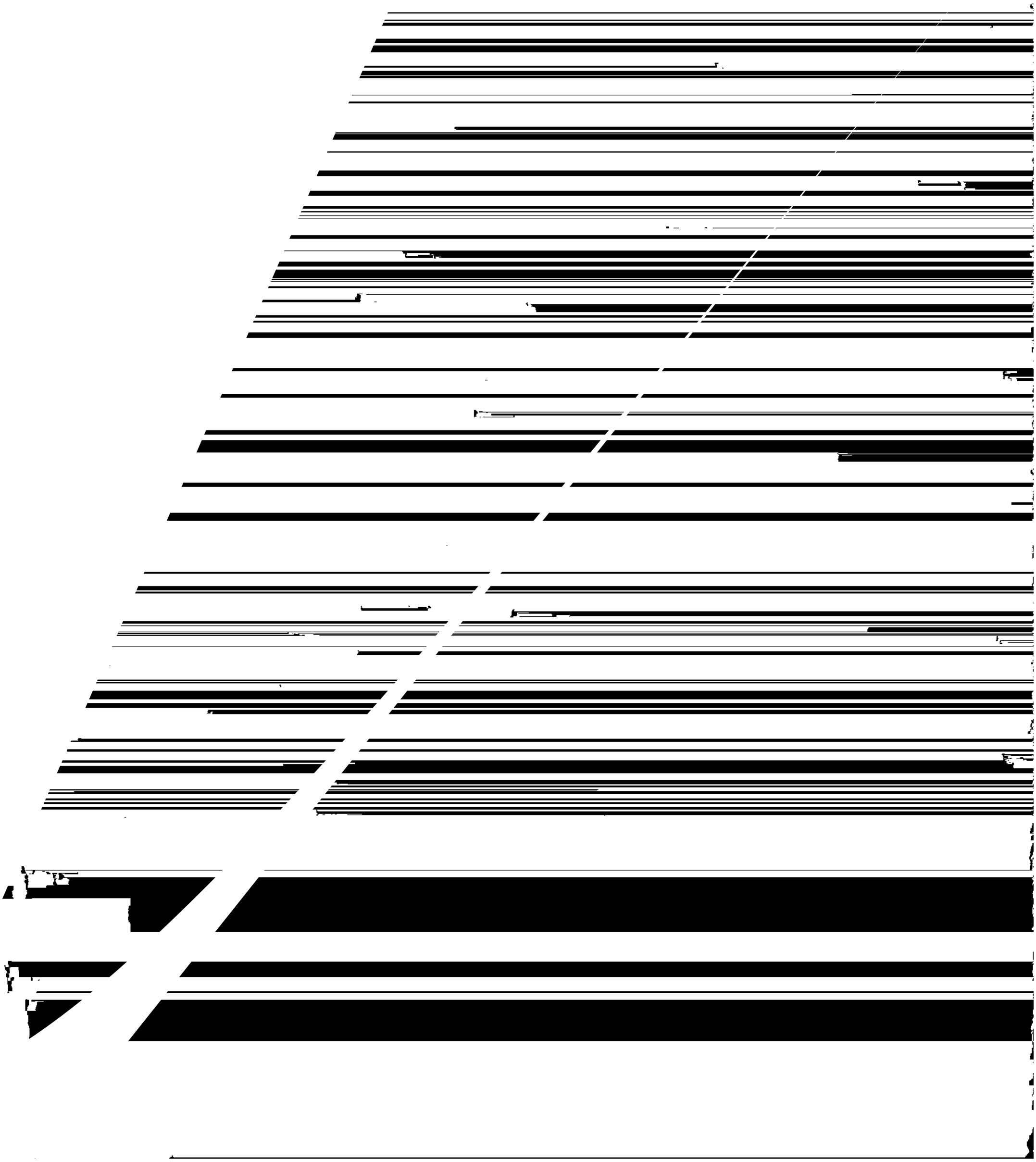


FIGURE 14.7 LM741 data sheet. *(Reprinted with permission from Fairchild Semiconductor and National Semiconductor.)*

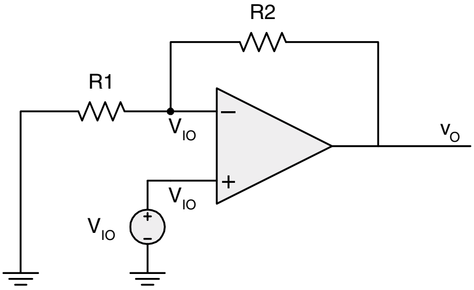


FIGURE 14.8 Analysis of input offset voltage.

the accuracy of the op-amp, the manufacturer will specify V_{IO} at room temperature (25°C) and at other temperature thresholds and provide coefficients or charts that relate the change in V_{IO} to temperature. Some op-amps, including the LM741, include compensation inputs that can be used to null-out the input offset error with external resistors. There is a limit to the practicality of this approach, however, resulting from temperature effects and the variation between components.

V_{IO} may not cause much trouble in an op-amp circuit with small gain, because a few millivolts of offset may be relatively insignificant as compared to several volts of actual signal. However, high-gain op-amp circuits that handle inputs with amplitudes in the millivolt range may be substantially degraded by V_{IO} effects. It may be desirable to AC-couple the op-amp's input when the frequencies of interest are high enough to make AC coupling feasible. The inverting circuit in Fig. 14.9 uses a capacitor to block a DC path to the op-amp's negative input. Because V_{IO} is a DC offset, the gain of the circuit is analyzed at DC to determine the amplification of V_{IO}. The impedance of the capacitor is ideally infinite at DC, resulting in unity gain for V_{IO}. For this circuit to function properly, the input frequency must be sufficiently high to not be attenuated by the highpass filter created by the series resistor and capacitor. The circuit exhibits a cutoff frequency where $f_C = 1 \div 2\pi C\text{R1}$. Therefore, frequencies much larger than f_C will be amplified by the idealized gain factor, –(R2 ÷ R1), when the capacitor essentially becomes a short circuit between the input and R1. The result is high gain at the frequencies of interest, yielding volts of output magnitude with only millivolts of V_{IO} error resulting from unity gain at DC.

Another parameter that relates to input offset voltage is the *power supply rejection ratio* (PSRR). The PSRR relates changes in the supply voltage to changes in V_{IO}. As the power rails fluctuate during normal operation, they influence the internal characteristics of the op-amp. PSRR is expressed in decibels as

$$PSRR = 20\log\frac{\Delta V_{CC}}{\Delta V_{IO}}$$

The LM741's minimum PSRR is specified at 77 dB. To ease arithmetic for the sake of discussion, we can round up to 80 dB and calculate PSRR = 10,000. This means that for every 1-V change in the supply voltage, V_{IO} changes by 100 µV. We know that V_{IO} is amplified by the gain of a DC-coupled circuit from the example in Fig. 14.8, indicating that a high-gain circuit is more susceptible to PSRR effects. PSRR declines with increasing frequency, making an op-amp more susceptible to power supply fluctuations.

Nonzero input currents are another source of inaccuracy in op-amp circuit, because real op-amps do not have infinite input resistance. Small nonzero currents flow into and out of the op-amp's in-

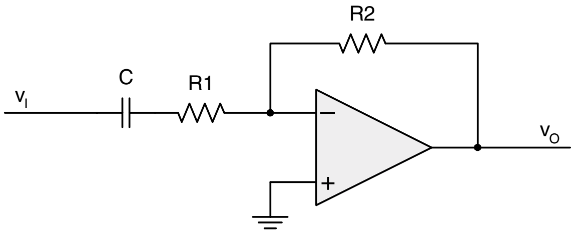
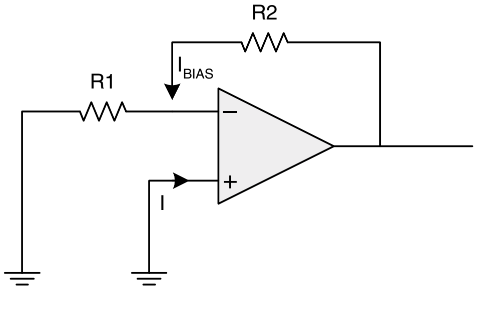


FIGURE 14.9 Mitigating V_{IO} with AC coupling.

puts. This current is the *input bias current*, I_{BIAS}. Ideal analyses of the preceding circuits assume that all currents flowing through R1 flow through R2 as well. This is clearly not true when the input impedance is finite. The result is that undesired voltage drops are created as input bias currents are drawn through resistors in an op-amp circuit. Consider the basic circuit in Fig. 14.10 with the inputs grounded to isolate I_{BIAS} effects independently from any input signal. The positive terminal remains at 0 V despite nonzero I_{BIAS}, because there is no resistance between it and ground (wire resistance is negligible). Therefore, the negative terminal is also at 0 V according to the virtual short assumption. While there is no current flowing through R1, because there is no voltage drop across it, I_{BIAS} flows from the output through R2. This results in a nonzero output voltage of $V_O = I_{BIAS}$R2 despite the fact that the circuit's inputs are grounded.

As seen in the LM741 data sheet, I_{BIAS} is measured in nanoamps for bipolar devices. CMOS op-amps specify I_{BIAS} in picoamps because of higher impedance MOSFET inputs. Practically speaking, many op-amp circuits with resistors measuring several kilohms or less do not have to worry about I_{BIAS} effects, because the product of nanoamps and resistance on the order of 10^3 Ω is on the order of microvolts. Of course, as circuit gains increase and allowable margins of error decrease, I_{BIAS} effects start to cause trouble. The problem is compounded by the fact that I_{BIAS} is not identical for each op-amp input as a result of slight physical variations in the circuitry associated with each input. A specification called *input offset current*, I_{IO}, is the difference between the two input bias currents. As seen in the LM741 data sheet, I_{IO} is smaller than I_{BIAS}, making I_{BIAS} the more troublesome characteristic.

Finite input bias current effects can be minimized by matching the induced offset voltages developed at each input. In Fig. 14.10, I_{BIAS} flowing into the positive terminal does not cause a voltage drop, which forces a corresponding I_{BIAS} to flow through R2 and create a nonzero output voltage. This circuit is augmented as shown in Fig. 14.11 by adding a resistor, R3, to the positive terminal.



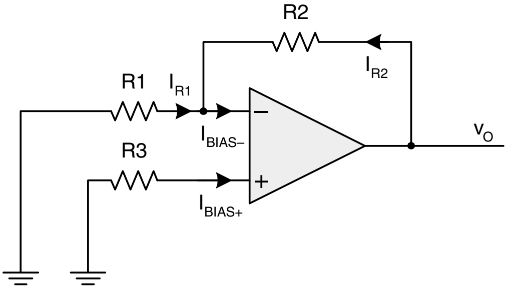


FIGURE 14.11 Mitigating I_{BIAS} with matched input resistor.

I_{BIAS+} now causes a voltage drop across $R3$, which is reflected at the negative terminal by reason of the virtual short. $V_- = V_+ = 0 - I_{BIAS+}R3$. Therefore, the current through $R1$ is $I_{R1} = I_{BIAS+}R3 \div R1$. The currents flowing through $R1$ and $R2$ into the negative terminal must equal I_{BIAS-}, because current does not simply disappear. Per node analysis, the sum of the currents entering a node must equal the sum of the currents leaving that node. This relationship can be restated to solve for the current through $R2$.

$$I_{R2} = I_{BIAS-} - I_{R1} = I_{BIAS-} - I_{BIAS+}\frac{R3}{R1}$$

Knowing I_{R2} enables the final expression of the output voltage,

$$V_O = V_- + I_{R2}R2 = (-I_{BIAS+})R3 + \left[I_{BIAS-} - I_{BIAS+}\frac{R3}{R1}\right]R2$$

By temporarily assuming that $I_{IO} = 0$, $I_{BIAS+} = I_{BIAS-}$. This reduces the output voltage expression to

$$V_O = I_{BIAS}\left[-R3 + R2 - \frac{R2R3}{R1}\right] = I_{BIAS}\left[R2 - R3\left(1 + \frac{R2}{R1}\right)\right]$$

This expression shows that the effects of I_{BIAS} can be compensated for by choosing $R3$ such that $V_O = 0$. Solving for $R3$ when $V_O = 0$ yields the parallel combination of $R1$ and $R2$,

$$R3 = \frac{R2}{1 + \frac{R2}{R1}} = \frac{R1R2}{R1 + R2}$$

A similar compensation can be designed for the case of an AC-coupled circuit, but it differs by taking into account the fact that no current flows through the coupling capacitor at DC. The circuit in Fig. 14.9 can be augmented by adding a resistor to the positive input terminal as shown in Fig. 14.12.

Unlike in the previous DC circuit, all of I_{BIAS-} flows through the feedback resistor, and the output voltage is

$$V_O = V_- + I_{BIAS-}R2 = V_+ + I_{BIAS-}R2 = -I_{BIAS+}R3 + I_{BIAS-}R2$$

modeled as the combination of an ideal op-amp with an offset voltage caused by finite CMRR, v_{CMRR}, as shown in Fig. 14.14. An ideal op-amp multiplies the input voltage, v_{CMRR}, by its differential gain, A, to yield an output voltage. Likewise, the nonideal op-amp being modeled multiplies the common-mode input voltage, $v_{CM} \approx v_D \approx v_I$, by its common mode gain to yield an output voltage. Therefore, $Av_{CMRR} = A_{CM}v_I$. This equivalency can be restated as $v_{CMRR} = A_{CM}v_I \div A$, which is actually a function of CMRR: $v_{CMRR} = v_I \div CMRR$.

With CMRR now modeled as an input that is a function of CMRR and the actual input voltage, the ideal op-amp model can be treated as a block to which the actual input signal is presented as shown in Fig. 14.15.

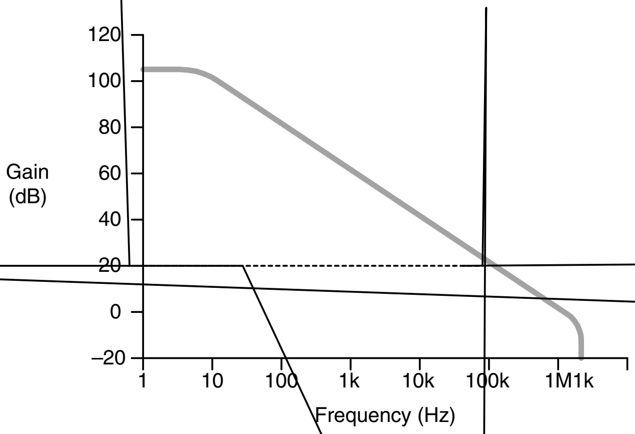
The total output voltage is the sum of the input voltage and $\mathrm{v}_{\mathrm{CMRR}}$ passed through the ideal op-amp as separate terms,

14.3 BANDWIDTH LIMITATIONS

An op-amp's ability to amplify a signal is a direct function of the frequencies involved. The maximum frequency at which an op-amp provides useful performance can be characterized in multiple ways. Manufacturers provide a variety of related specifications to assist in this task. First, there is the *gain-bandwidth product* (GBW), also referred to as *bandwidth* (BW). As the name implies, GBW specifies a gain at a certain frequency. As the frequency of operation is decreased, the gain increases, and vice versa. Most op-amp data sheets provide either a bandwidth number, sometimes specified as the unity-gain bandwidth, or a chart that relates gain to frequency. The large-signal voltage gain of the LM741 remains relatively constant for frequencies only under 10 Hz (not kHz!). It is called "large-signal," because the response is essentially at DC; "small-signal" would denote practical operating frequencies that are typically orders of magnitude higher. Near 10 Hz, the gain rolls off at the rate of –20 dB per decade and reaches unity gain, 0 dB, near 1 MHz. Above 1 MHz, the LM741's gain rapidly drops off. Keep in mind, however, that this is the LM741's open-loop response. When a closed-loop circuit is formed, the gain is substantially lowered, thereby increasing the circuit's usable bandwidth over which the gain remains constant.

Figure 14.16 shows an approximate gain versus frequency curve for an older bipolar op-amp such as the LM741. The gain of the open-loop op-amp is approximately 100 dB below 10 Hz. Observe that, if the closed-loop gain is set at 10 (20 dB), the constant gain portion of the curve drops by approximately 80 dB, which correlates to four decades of frequency. Therefore, the closed-loop bandwidth increases to 10 Hz multiplied by four decades, or 100 kHz. Put another way, 100 dB equals 100,000. The gain-bandwidth product is approximately 100,000 × 10 Hz = 1 MHz. If a closed-loop gain of 10 is used, the bandwidth is 1 MHz ÷ 10 = 100 kHz.

Two other interrelated frequency metrics are *slew rate* and *full-power bandwidth*. These specifications enable an engineer to determine whether a signal will be distorted by the op-amp based on the signal's frequency and amplitude. Slew rate defines the rate at which the op-amp can change its output voltage. The LM741's specified slew rate is 0.5 V/µs. If the desired output signal has a component that changes voltage faster than the slew rate, the op-amp will not be able to fully reproduce the signal and is said to be *slew-rate limited*. An example of slew-rate limiting is shown in Fig. 14.17, where the desired sine wave output, a pure signal with a single frequency, is converted to a triangle waveform that transitions at the op-amp's maximum slew rate.



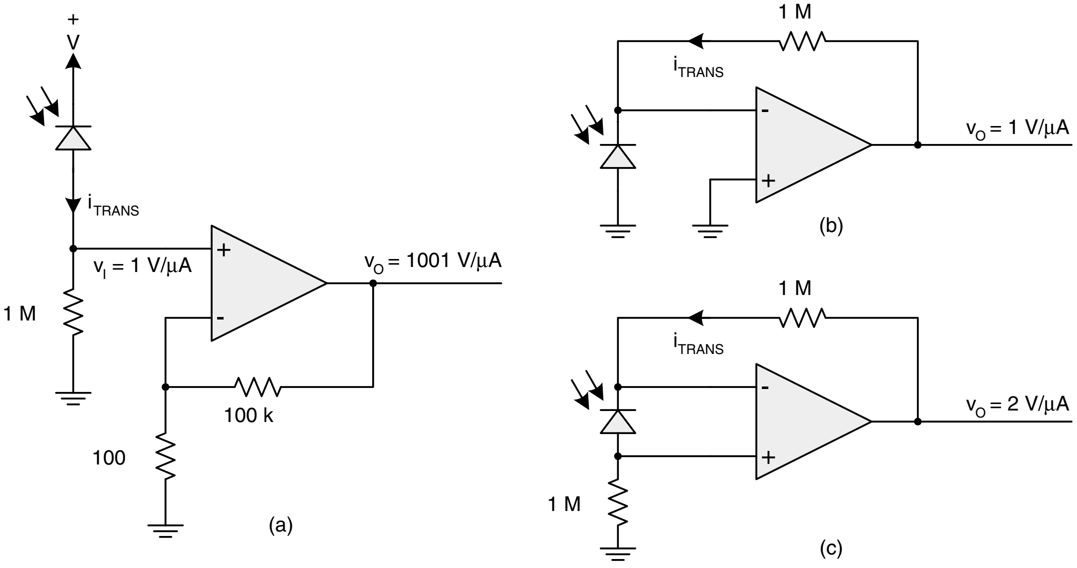


FIGURE 14.19 Amplification of very weak transducer signals.

A signal applied to a noninverting circuit directly drives the op-amp's positive input and, in doing so, establishes a voltage at both the positive and negative inputs due to the virtual short. Likewise, an inverting circuit's op-amp input voltages are established by the bias applied to the positive terminal. Referring back to the standard inverting circuit in Fig. 14.5, a voltage drop is developed across the input resistor, R1, that sets the input current: $i_{IN} = (v_{IN} - v_{\_}) \div R1 = v_{IN} \div R1$. Therefore, the input resistance seen by the signal is equal to this series input resistor. This basic circuit places a practical ceiling on the input resistance, because the circuit's gain is inversely proportional to that input resistor. If this ceiling is reached, and still higher input resistance is necessary, the basic inverting circuit can be augmented as shown in Fig. 14.20 with a "tee" topology in the feedback path.

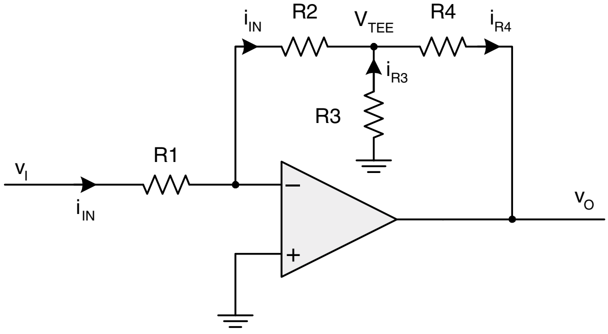


FIGURE 14.20 Inverting circuit with higher input resistance using tee feedback topology.

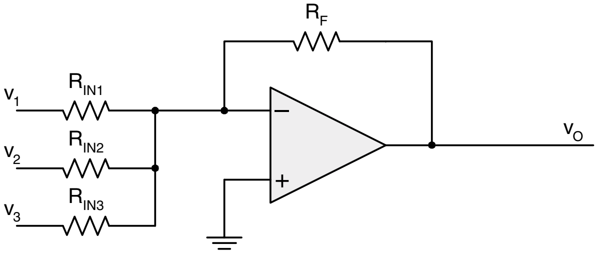


FIGURE 14.22 Summation circuit.

inverting topology. Each input is unaffected by the others, because the negative terminal is always at 0 V by reason of the virtual short. Therefore, each input resistor contributes an input current component determined by Ohm's law. These currents add and collectively pass through the feedback resistor to create an output voltage.

A weighted summation can be designed by individually selecting input resistors according to the application requirements. If the input resistors are all equal, the gain ratios for each input signal are equal, and the summation is balanced.

$$v_O = -\frac{R_F}{R_{IN}}[v_1 + v_2 + v_3]$$

Similarly, a noninverting circuit can be used to combine multiple input voltages without the −1 factor as shown in Fig. 14.23. Although usually referred to as a *noninverting summer,* this is more of an averaging circuit than a summer. The input voltages are averaged by the input resistor network, and this average level is then multiplied by the gain of the noninverting circuit.

Analyzing the input resistor network can be difficult without a common circuit-analysis trick that relies on the principle of *superposition*. Superposition works with a linear transfer function that relates an output to multiple inputs multiplied by some gain factor. For example, two signals, V_1 and V_2, add together and are multiplied to yield an output voltage: $V_O = A(V_1 + V_2)$. By the principle of superposition, the input terms can be broken up, computed separately, and then recombined at the output to yield a final expression: $V_{O1} = AV_1$ and $V_{O2} = AV_2$, thus $V_O = V_{O1} + V_{O2} = A(V_1 + V_2)$. Superposition works only when the transfer function is linear.

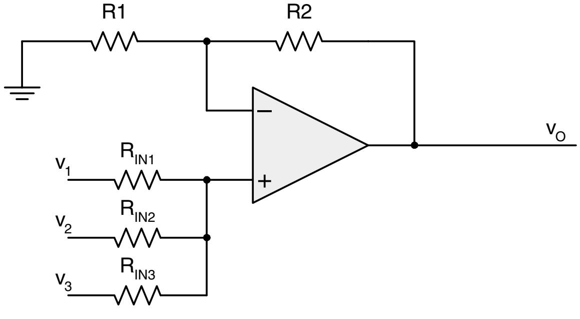
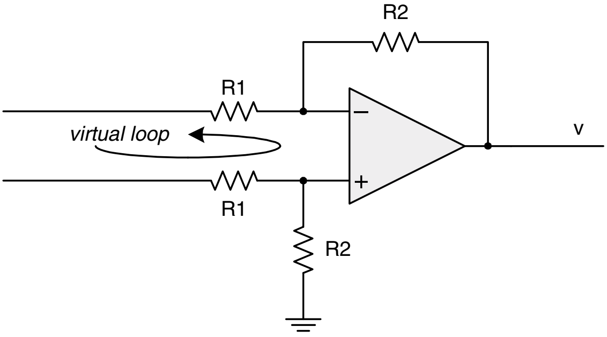


FIGURE 14.23 Noninverting summation circuit.

In circuit analysis terms, a single input voltage can be isolated from the others by tying each other input to ground and then evaluating the circuit as if that single input voltage were the only input present. This analysis method can be applied to each input in succession, and then the partial results can be summed to yield a final output expression. Superposition is applied to analyze the noninverting summer as follows. First, v_2 and v_3 are grounded, yielding a basic voltage divider expression for the op-amp input voltage that translates to a partial output voltage.

To ease calculation for the sake of discussion, assume that $R1 = R2 = R_{IN1} = R_{IN2} = R_{IN3} = 10\ \text{k}\Omega$. Under these conditions,

This same treatment is performed separately for v_2



By superposition, the linear partial terms, v_{O+} and v_{O-}, are summed to yield a final output expression that clearly shows that this circuit is a difference amplifier.

A drawback of the single op-amp difference amplifier is that it has a relatively low input resistance. Using the virtual short concept, the two op-amp terminals are at the same voltage, thereby creating a virtual loop circuit consisting of the differential voltage input and the two input resistors, R1. Because this virtual circuit consists only of the input voltage source and the two resistors, the input resistance is observed to be equal to 2R1 by inspection. As with the basic inverting op-amp circuit, there is a practical ceiling imposed on input resistance caused by the circuit's gain and the range of resistances that are practical to use in a real circuit.

Many applications in which a difference amplifier is necessary involve weak signal sources such as an unbuffered transducer. To solve this problem, a more complex difference amplifier can be constructed with multiple op-amps to present a much higher input resistance. Usually called *instrumentation amplifiers*, these circuits commonly consist of three op-amps, two of which are configured in the noninverting topology for very high input resistance. The third op-amp is configured in the just-mentioned difference amplification topology. As with the example in Fig. 14.18, the noninverting op-amps buffer each half of the differential input signal, and the second op-amp stage performs the final difference function. If such a circuit is required in a digital system, it may be most practical to use an integrated instrumentation amplifier as manufactured by such companies as Linear Technology (e.g., LT1167) and Texas Instruments (e.g., INA332) rather than constructing one from discrete op-amps.

14.6 ACTIVE FILTERS

Active filters perform the same basic frequency passing and blocking function as passive filters, but they can simultaneously amplify the signal to form a filter that has unity or higher gain. This is in contrast to passive filters that achieve less than unity gain because of finite losses inherent in the components from which they are constructed. Op-amps can be used to implement active filters as long as their gain-bandwidth product is not exceeded. Figure 14.25 shows familiar first-order low-

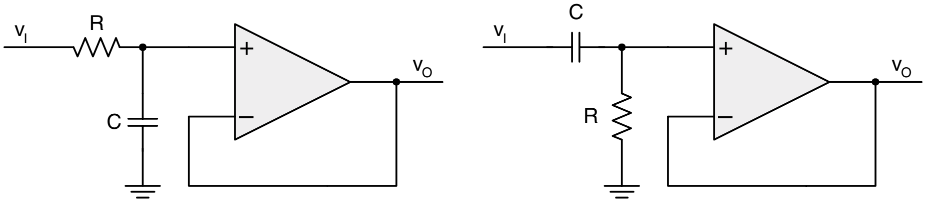


FIGURE 14.25 First-order active filters.

pass and highpass active filters implemented with op-amps. These simple filters buffer a passive filter with a noninverting op-amp stage. In this example, the configurations are for unity gain, although higher gains are possible. Because of the high input resistance of an op-amp, there is little signal loss through the series elements while operating in the passband. Unlike a passive filter whose characteristics are influenced by the load being driven, the op-amp isolates the load from the filter elements.

Filter design is a topic in electrical engineering that can get quite complex when very specific and demanding frequency response characteristics are necessary. Active filters add to this complexity as a result of nonideal op-amp characteristics. Although a complete discussion of filter design is outside the scope of this text, certain filtering tasks can be accomplished by drawing on a basic familiarity with common circuits. A common second-order topology used to implement active filters is the Sallen-Key filter. Sallen-Key lowpass and highpass filters are implemented with two resistors and two capacitors each for unity gain in the passband, as shown in Fig. 14.26. If higher gains are desired, two resistors can be added per the standard noninverting amplifier circuit topology. As with a passive second-order filter, the frequency response curve falls off at 40 dB per decade beyond the cut-off frequency.

The Sallen-Key lowpass filter operates by shunting the op-amp's input path to low-impedance sources at high frequencies. C1 shunts the signal to ground as in a passive filter, and C2 shunts the signal to the op-amp's output. The highpass filter operates in the reverse manner. At low frequencies, C1 blocks the incoming signal and allows R2 to feed the output back to the input. Simultaneously, C2 blocks the signal, which pulls the input to ground. The general expression for the cut-off frequency, f_C, is

$$f_C = \frac{1}{2\pi\sqrt{R1R2C1C2}}$$

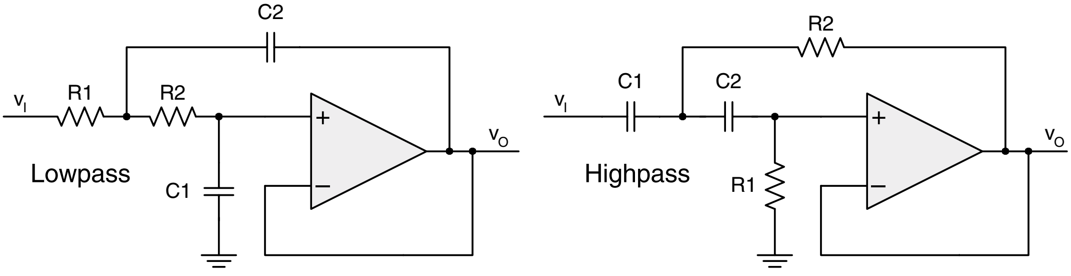
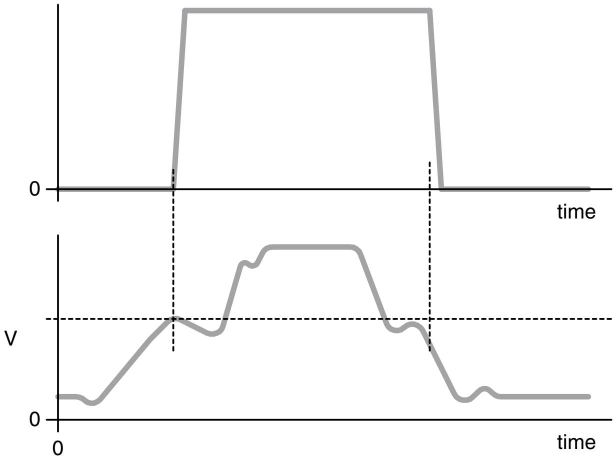


FIGURE 14.26 Sallen-Key second-order active filters.

It can be difficult to achieve a perfectly monotonic signal because of ambient noise in a system. If the signal can be guaranteed to rise and fall quickly, the window of opportunity for noise to trigger an undesired response at V_{REF} is limited. This is how the majority of signals in a digital system operate. The rise and fall times are relatively fast, and the signals remain stable at logic-high and logic-low voltages. Problems do arise, however, when excessive noise or other signal integrity issues manifest themselves by causing nonmonotonic signal transitions. In a purely digital context, such problems can be solved with proper engineering solutions to reduce and shield noise and signal integrity problems. Most real-world analog signals do not behave in a clean binary fashion, which is why they often require analog circuits including op-amps and comparators to properly interface with digital systems.

Threshold comparison can be improved by adding *hysteresis* to an otherwise open-loop voltage comparator. Hysteresis is the application of two thresholds to stabilize a comparator so that it does not change its state with minute changes in the input voltage. Stabilization is desirable in situations wherein the applied voltage hovers near the threshold voltage for more than a brief span of time as seen in Fig. 14.29. Rather than a single threshold, separate low-to-high, V_{TLH}, and high-to-low, V_{THL}, thresholds are designed. V_{TLH} is higher than V_{THL}, as demonstrated in Fig. 14.30, where a much cleaner output is obtained as compared to the previous case. Note that the hysteresis created by the two thresholds prevents the comparator's output from returning to logic 0 when the input declines slightly after triggering a logic-1 output. Similarly, the input's nonmonotonic falling-edge does not cause the output to bounce, because the hysteresis is chosen to be greater than the local perturbation.

A *hysteresis loop* is a common means of representing the two distinct thresholds governing the input/output transfer function. Figure 14.31 shows that, when the input is starting from the low side, the high threshold is used to trigger an output state transition. As soon as the input crosses the high threshold, the output goes high, and the low threshold is now applied to the input comparison. Notice the advantage here. Once the input rises above the high threshold, a lower threshold is instantly substituted. This means that, if the input signal wanders and declines slightly, it is still above the lower threshold, and the output is unaffected. For the output to return low, the input must now fall below the low threshold. And as soon as this occurs, the high threshold is again activated so that the input must rise significantly before an output stage change will occur.



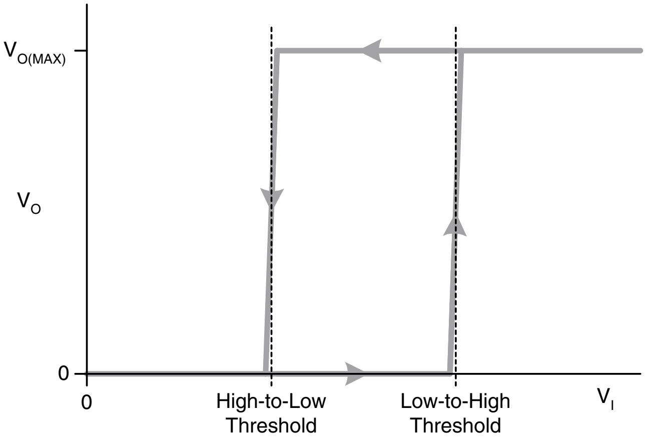


FIGURE 14.31 Hysteresis loop.

The degree of hysteresis designed into a comparator circuit is determined by the difference between the high and low thresholds. A small difference is less tolerant of noise. However, a larger difference has a more muted response, which must be considered in its effects on a system's behavior. Dual comparison thresholds that create hysteresis are implemented by applying positive feedback rather than negative feedback to an op-amp or comparator. Such a circuit is shown in Fig. 14.32. The circuit looks very similar to a conventional closed-loop amplifier, but the feedback is applied to the positive terminal rather than the negative terminal.

The positive feedback through R2 results in a voltage at the positive terminal that is determined by the basic voltage divider expression,

$$v_+ = v_O + (v_I - v_O)\frac{R2}{R1+R2} = v_O\left(1 - \frac{R2}{R1+R2}\right) + v_I\frac{R2}{R1+R2} = v_O\frac{R1}{R1+R2} + v_I\frac{R2}{R1+R2}$$

Therefore, the output pulls v$_+$ down when it is low and pulls v$_+$ up when it is high. This means that, if v$_I$ is increasing and trying to switch the comparator state from low to high, it must be raised to a higher voltage threshold to counteract the pull-down effect when the comparator's output is al-

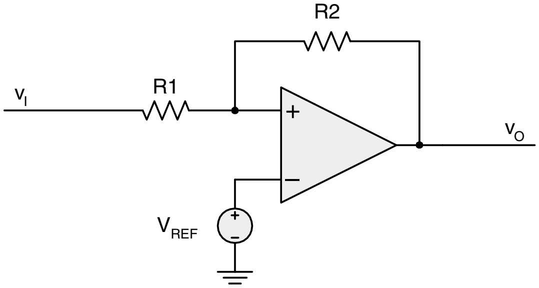


FIGURE 14.32 Hysteresis created by positive feedback.

CHAPTER 15
Analog Interfaces for Digital Systems

The intersection of analog and digital worlds has given rise to a tremendously broad range of applications for digital systems. Digital cellular telephones, enhanced radar systems, and computerized engine controls are just a few modern benefits enabled by data conversion circuits. Analog-to-digital and digital-to-analog converters enable computers to interact with the real world by representing continuous analog signals as sequences of discrete numbers.

The first portion of this chapter discusses topics including quantization, sampling rate, and the Nyquist frequency. These concepts form the foundation of data conversion and enable an engineer to evaluate the conversion requirements for individual applications. Specific analog-to-digital and digital-to-analog conversion techniques are presented in the next portion of the chapter. Selecting the correct data conversion IC is a combination of identifying the application requirements and then finding a device that matches these specifications.

Following the initial evaluation process, designing an effective data conversion circuit at the system level varies in complexity with how accurate the conversion must be. Some applications with moderate accuracy requirements can be implemented without much difficulty by following the manufacturer's recommended connection diagrams in their data sheets and application notes. More complex conversion circuits, such as those in digital radio transceivers or high-accuracy instruments, can present significant analog design tasks wherein noise reduction and stability over time and temperature are key challenges. While briefly discussed at the end of this chapter, these high-end applications require further reading into sampling theory and more advanced analog design skills.

15.1 CONVERSION BETWEEN ANALOG AND DIGITAL DOMAINS

Many digital systems interact with their environment by measuring incoming analog signals, such as sound from a microphone, and emitting other analog signals that have been processed in some manner, such as playing a CD on your computer's speakers. These functions are not natural to digital systems, because a binary signal can have only two discrete states, 1 and 0, whereas analog signals

spaced sampling intervals are shown at an arbitrary phase offset from the sine wave to illustrate the arbitrary alignment of samples to the incoming signal.

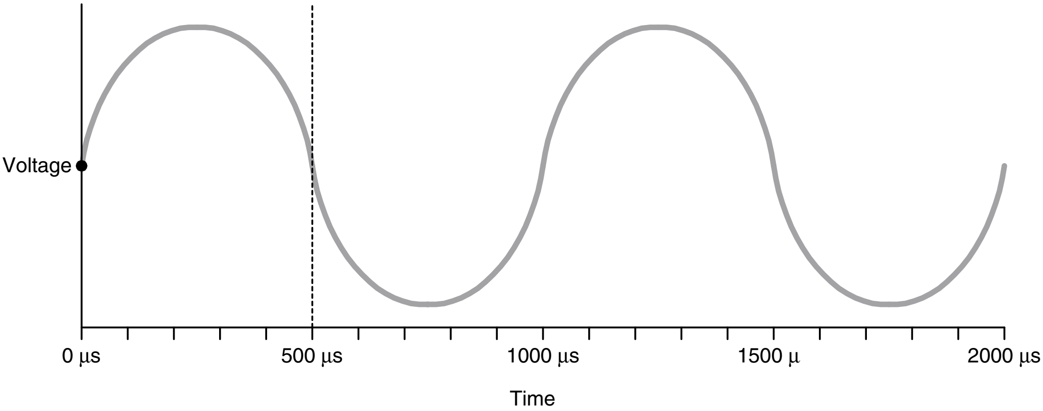
Discrete digital samples are reconstructed to generate an analog output that is a facsimile of the original analog input minus a finite degree of distortion. Figure 15.4 shows the 1-kHz sine wave samples as they are emitted from a DAC. On its own, the sampling process creates a stair-step output that bears similarity to the original signal but is substantially distorted. The output is not a pure sine wave and contains added high-frequency components at the sampling frequency—10 kHz in this case. This undesirable stair-step output must be passed through a lowpass filter to more closely reconstruct the original signal. A lowpass filter converts the sharp, high-frequency edges of the DAC output into smoother transitions that approximate the original signal. The output will never be identical to the original signal, but a combination of sufficient sampling rate and proper filter design can come very close.

As the ratio of sampling rate to signal frequency decreases, the conversion accuracy becomes more coarse. Figure 15.5 shows the same 1-kHz sine wave being sampled at only three times the signal frequency, or 3 kHz. Consider how the stair-step output of a DAC looks with this sampling scheme. Here, a filter with sharper roll-off is required, because the undesirable frequency components of the DAC output are spaced only 2 kHz apart from the frequency of interest instead of nearly a decade as in the previous example. A suitable filter would be able to output a nearly pure 1-kHz sine wave, but the signal's amplitude would not match that of the input, because the maximum DAC output voltage would be less than the peak value of the input signal, as indicated by the position of the samples as shown.

Preservation of a signal's amplitude is not critical, because a signal can always be amplified. It is critical, however, to preserve the frequency components of a signal because, once lost, they can never be recovered. Basic sampling theory was pioneered by Harry Nyquist, a twentieth century mathematician who worked for Bell Labs. Nyquist's theorem states that a signal must be sampled at greater than twice its highest frequency to enable the preservation of its informational, or frequency, content. Nyquists's theorem assumes a uniform sampling interval, which is the manner in which most data conversion mechanisms operate. The *Nyquist frequency* is a common term that refers to one-half of the sampling frequency. A data conversion system is said to operate below the Nyquist frequency when the applied signal's highest frequency is less than half the sampling frequency.

Nyquist's theorem can be understood from a qualitative perspective by considering operation at exactly the Nyquist frequency. Figure 15.6 shows a 1-kHz sine wave being sampled at 2 kHz. There are a range of possible phase differences between the sampling interval and the signal itself. The worst-case alignment is shown wherein the samples coincide with the sine wave's zero crossing. Because each sample is identical, the observed result is simply a DC voltage.

If the sampling frequency is increased by a small amount so that the signal is less than the Nyquist frequency, it is impossible for consecutive samples to align themselves at the signal's zero crossings. The amplitude measurement may suffer substantially, but the basic information content of the signal—that it is a 1-kHz sine wave—will not be lost. As the prior examples show, operating with substantially higher sampling rates, or operating at substantially below the Nyquist frequency, increases the accuracy of the data conversion process.



When a data conversion circuit is operated above the Nyquist frequency, *aliasing* develops, which distorts the information content of a signal. Aliasing occurs when a high-frequency signal is made to appear as a low-frequency signal as a result of inadequate sampling. Assume that a 1-kHz input is expected and a 3-kHz sampling rate is designed to satisfy the Nyquist theorem with some margin. For some reason, a 2-kHz signal enters the system. Figure 15.7 shows how this signal might be sampled. The 333 μs sampling interval is too slow to capture each 250 μs high and low phase of the 2-kHz signal. As a result, the samples convey information that is drastically different from the input. Instead of 2 kHz, the high-low-high interval of the samples indicates a 1-kHz signal! This 1-kHz aliased signal is the difference between the sampling rate and the actual signal frequency.

Aliasing applies to the analog-to-digital conversion process. When converting from digital to analog, the DAC is inherently self-limited by the sampling rate such that the highest frequency it can generate is half the sampling rate, or the Nyquist frequency. This behavior would result if consecu-

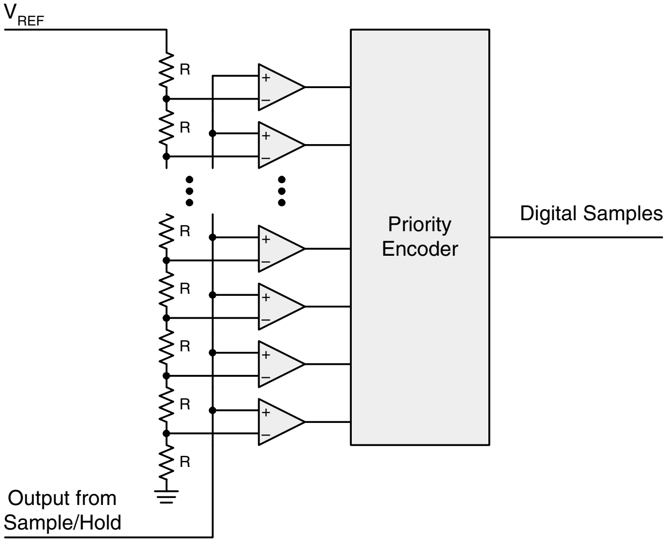


FIGURE 15.10 Flash ADC circuit.

requires 4,095 parallel comparators. When a voltage is applied to the flash circuit, one or more comparators may emit a logic 1. A priority encoder generates the final N-bit output based on the highest comparator that is emitting a logic 1. This circuit is called a flash ADC, because it is very fast; an analog input is converted to a digital sample in one step.

Flash ADCs are fast, but their complexity doubles with each added bit of resolution. Such ADCs are available with maximum sampling rates over 100 MHz and resolutions between 8 and 16 bits from manufacturers including Analog Devices, Intersil, National Semiconductor, and Texas Instruments.

When very high sampling rates are not necessary, alternative ADC circuits are able to accomplish the task with lower cost and increased resolution. High-quality audio applications commonly use 24-bit ADCs with sampling rates of either 48 or 96 kHz. Below 16 bits of resolution and 20 kHz, many inexpensive and low-power ADCs are available. successive-approximation and sigma-delta ADCs are manufactured by the same companies that make flash ADCs. In addition, Crystal Semiconductor offers a line of ADCs optimized for digital audio applications.

A successive-approximation ADC uses an internal DAC/comparator feedback loop to home in on the digital code that corresponds to the applied analog input. Figure 15.11 shows this feedback loop along with control logic that varies the code until the DAC output matches the input. Relatively dumb control logic could simply increment the code starting from 0 until the comparator's output changed from high to low. This would mean that an N-bit ADC would require up to 2^N cycles to perform a conversion. Instead, a successive-approximation ADC performs a binary search to accomplish the same task in only N cycles. A digital code of 0 is used as a starting point. Each bit in the code, starting from the most significant bit, is set, and the comparator's output is tested each time. If the output is low, the DAC voltage exceeds the input and, therefore, the bit that was set should be cleared. Otherwise, the bit is left set.

To illustrate how a successive-approximation ADC functions, consider an 8-bit ADC with a range from 0 to 5 V and an input level of 3 V. Each conversion quantum is 19.61 mV. Table 15.1 lists the eight sequential steps in performing the data conversion. In reality, the hypothetical ADC circuit may output 0x98 or 0x99 with a 3-V input, depending on the ambient electrical and thermal conditions. When an input is on the border between two quanta, slight changes in supply voltage, noise, and temperature can skew the result up or down by one digital code. The final result is chosen as 0x98, because the next code, 0x99, corresponds to a voltage that is slightly higher than the input voltage. This gets back to the concept of conversion accuracy. Manufacturers specify ADCs with worst-case accuracies. Additionally, the parameters of the circuit into which they are designed can further degrade the conversion accuracy.

TABLE 15.1 Eight-Bit Successive-Approximation Conversion Steps

Cycle	Test Code	DAC Voltage	Comparator Output	Resultant Code
1	**1**0000000	2.51	1	**1**0000000
2	1**1**000000	3.77	0	1**0**000000
3	10**1**00000	3.14	0	10**0**00000
4	100**1**0000	2.82	1	100**1**0000
5	1001**1**000	2.98	1	1001**1**000
6	10011**1**00	3.10	0	10011**0**00
7	100110**1**0	3.02	0	100110**0**0
8	1001100**1**	3.00	0	1001100**0**

A sigma-delta ADC over-samples the input at very coarse resolution: one bit per sample! To create a high-resolution sample, a typical sigma-delta ADC oversamples by 128 or 256 times the nominal sampling frequency and then passes the serial samples through a digital filter to create a usable set of N-bit samples at the nominal sampling frequency. The basic theory behind a sigma-delta ADC has been around for a long time, but its practical implementation is more recent because of its reliance on digital filter logic, which is now inexpensive to manufacture on an IC. Figure 15.12 shows a sigma-delta ADC incorporating a voltage summation stage, an integrator, a comparator, a 1-bit DAC, and a digital filter. The summation stage subtracts the DAC output from the input voltage. The integrator is a circuit that accumulates the resulting sum over time.

For a given input, the sigma-delta circuit will emit a serial set of samples with an average DC value over time that equals the input voltage. The integrator keeps track of the difference between the input and the DAC feedback voltage. When the comparator sees that this running difference exceeds 0, it causes a negative feedback through the DAC and summation stage. When the difference is

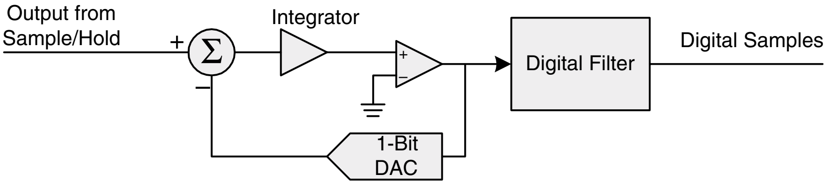


FIGURE 15.12 Sigma-delta ADC circuit.

less than 0, the comparator's output is 0, and there is no negative feedback, causing the running difference to increase once again.

The disadvantage of a sigma-delta ADC is that it requires a high oversampling frequency to function. However, this is not a problem at low frequencies, such as audio, where these converters are commonly employed. A sigma-delta ADC is able to deliver very high resolutions (e.g., 24 bits) with high accuracy, because most of its complexity is in the digital filter, and only a coarse single-bit conversion is performed. This means that the circuit is less susceptible to noise. Digital logic is much more tolerant of ambient noise as compared to delicate analog comparators and amplifiers.

A key advantage of sigma-delta ADCs for the system designer is that expensive lowpass filters with sharp roll-offs are not required. Since the actual sampling frequency is so much higher than the signal's frequency content, an inexpensive single-pole RC lowpass filter is generally sufficient. Consider a CD audio sampling application in which the maximum input frequency is 20 kHz and the nominal sampling rate is 44.1 kHz. A sigma-delta ADC might sample this signal at 128 × 44.1 kHz = 5.6448 MHz. Therefore, the Nyquist frequency is raised to approximately 2.8 MHz from 22 kHz! A first-order filter with f_C = 20 kHz would attenuate potentially aliasing frequencies by more than 40 dB. In contrast, a normal ADC would require a much more costly filter to provide the same attenuation where the passband and stopband are separated by only 2 kHz.

15.4 DAC CIRCUITS

Unlike an ADC, a DAC does not require a sample and hold circuit, because the instantaneous sample events are driven from the discrete digital domain where each clock cycle activates a new sample. A DAC consists of a digital interface and the conversion circuit. Two of the most common types of conversion circuits are the *R-2R ladder* and sigma-delta designs.

The R-2R ladder DAC uses the concept of current summation as found in an inverting op-amp summing circuit. Two resistance values, R and 2R, are connected in a multistage network as shown in Fig. 15.13 (using a four-bit example for the sake of brevity). This circuit is best analyzed using superposition: set one input bit to logic 1 (V_{REF}) and the others to logic 0 (ground). When this is done, the resistor ladder can be quickly simplified by combining parallel and series resistances, because all nodes other than the logic 1 input are at 0 V. Knowing that the voltage at the op-amp's negative terminal is also 0, the current through the resistor ladder can be determined and, therefore, the output voltage can be calculated.

After calculating the partial output voltage due to each individual input, the following overall expression for V_O is obtained:

$$V_O = -V_{REF}\left[\frac{D3}{2} + \frac{D2}{4} + \frac{D1}{8} + \frac{D0}{16}\right]$$

This expression allows linear control of V_O ranging from 0 to within one least-significant bit position (1/16 in this case) of $-V_{REF}$. Similar results can be obtained with the basic op-amp summer cirquencies generated. These devices are commonly used in digital audio equipment because of their low overall system cost and high resolution (16 to 24 bits) for frequencies below 100 kHz.

15.5 FILTERS IN DATA CONVERSION SYSTEMS

General noise-reduction filters are typically found on the power supplies feeding data conversion circuits, because less distortion of the analog signals results when there is less overall noise in the system. Purely digital systems are not immune to noise, but their tolerance threshold is much higher than for analog systems because of their highly quantized binary signals. It takes noise of greater magnitude to turn a 0 into a 1 than it does to distort a continuous analog signal.

In addition to general noise reduction, anti-aliasing filters are a key design aspect of data conversion systems. Filtering requirements dictated by the sampling rate and by the presence of undesired frequency content can be quite stringent. As the gap between the Nyquist frequency and the undesired frequencies decreases, more complex filters are necessary. The design of such complex filters requires a substantial set of analog design skills, the majority of which are outside the scope of this book. However, this chapter closes by identifying some of the issues that arise in anti-alias filtering so that you may be aware of them.

The first step in specifying an anti-aliasing filter is to identify how much attenuation is necessary in the stopband. An ADC or DAC can resolve an analog signal only to a finite resolution given by the number of bits that are supported. It is therefore unnecessary to attenuate high-frequency content beyond the point at which the circuit's inherent capabilities reach their limit. Attenuating unwanted signals to less than one-half of a voltage quantum renders them statistically insignificant. If a system represents an analog signal with N bits of resolution, this minimum attenuation, A_{MIN}, is given as

One-half of a quantum is represented by the added power-of-two beyond that specified by the conversion resolution. The term represents an allowance for the average quantization noise magnitude. As expected, a higher-resolution conversion requires greater attenuation, because the

This page intentionally left blank.

P • A • R • T • 4

DIGITAL SYSTEM DESIGN IN PRACTICE

This page intentionally left blank.

CHAPTER 16
Clock Distribution

Clocks are inherently critical pieces of a digital system. Reliable operation requires the distribution of electrically clean, well timed clocks to all synchronous components in the system. Clocking problems are one of the last bugs that an engineer wants to have in a system, because everything else is built on the assumption of nearly ideal clocks. This chapter concentrates on the means of distributing low-skew and low-jitter clocks in a system. Most systems require the design of a *clock tree*—a circuit that uses an oscillator of some type to create a clock and then distributes that clock to multiple loads, akin to branches in a tree. Simple clock trees may have a single level of hierarchy in which the oscillator directly drives a few loads. More complex trees have several levels of buffers and other components when tens of loads are present.

Basic information on crystal oscillators is presented first to assist in the selection of a suitable time base from which to begin a clock tree. Once a master clock has been produced, low-skew buffers are the common means of replicating that clock to several loads. These buffers are explained with examples incorporating length matching for low-skew and termination resistors for signal integrity. Buffers are followed up with a discussion of phase-locked loops, commonly used to implement "zero-delay" buffers in clock trees. These devices become important when a system contains multiple boards and when there are special clocking sources other than a stand-alone oscillator. Low-skew and zero-delay buffers form the basis for most clock tree designs.

The second portion of the chapter discusses more advanced clocking concepts beginning with frequency synthesis. Originally conceived for analog and RF applications, frequency synthesis is an important part of many high-performance digital systems. It allows multiple clocks to be derived from a single time base and is how a leading-edge microprocessor operates many times faster internally than it does externally. Frequency synthesis can also be important when processing data between multiple interfaces that run at different frequencies.

Next, delay-locked loop technology is presented, because it can accomplish the same basic function as a phase-locked loop in many cases but is a purely digital circuit with resultant implementation advantages. The chapter concludes with a brief discussion of source-synchronous interfaces as a necessary alternative to conventional synchronous design when dealing with very high frequencies.

16.1 CRYSTAL OSCILLATORS AND CERAMIC RESONATORS

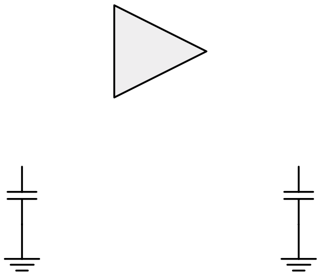
An electronic clock consists of an amplifier with a passive time-base element coupled into its feedback loop. It has previously been shown that a simple oscillator can be formed with just an RC time base and an inverter that serves as an amplifier. Simpler yet is connecting the output of the inverter directly to its input via a piece of wire sized to provide a certain time delay. While simple, neither of these approaches yields a sufficiently accurate clock source in most applications. Accurate timebases are created by exploiting the resonant properties of *piezoelectric* crystals cut to specific sizes. Piezoelectricity is the property found in certain crystals whereby slight changes in the crystalline structure result in a small electric field, and vice versa; exposing the crystal to an electric field causes slight deformation of its structure. Quartz is the most common piezoelectric crystal in use for oscillators, but other such crystals, both natural and synthetic, have this property. A solid object's natural resonant frequency is a function of its physical dimensions and its composition. A crystal such as quartz or certain ceramics has a predictable resonant frequency that can be finely adjusted by varying the size of the crystal slab. Furthermore, the resonant frequency is largely insensitive to variations in temperature and voltage.

A piezoelectric crystal of known resonant frequency can be incorporated with an inverting amplifier to yield an accurate clock. The amplifier drives one end of the crystal, and the other end feeds back to the amplifier's input. With proper circuit design, the crystal begins to resonate as it is driven and quickly settles into a continuous oscillation that both stimulates and is maintained by the amplifier. A generic crystal oscillator circuit found in many digital ICs is shown in Fig. 16.1. Note the crystal's graphical representation. The IC contains an internal crystal driver, which is a specialized inverter. Externally, a crystal and load capacitors are required. The load capacitors form an LC resonant circuit in concert with the crystal that is made to appear inductive. Manufacturers specify crystals with a particular load capacitance requirement for proper oscillation. The two capacitors are typically selected to be the same value, C. When this is done, the overall load capacitance, C_L, is 0.5 C plus any stray capacitance, C_S, in the circuit. Stray capacitance is often in the may range of several picofarads. A crystal specified with C_L = 18 pF might use 22-pF capacitors assuming $C_S \approx$ 7 pF.

The circuit shown is a digital oscillator, because it emits a square wave binary signal. Analog applications such as RF use a linear amplifier to drive a sine wave instead.

It is rare to find a circumstance these days in which an engineer must design a digital oscillator from scratch. Many embedded microprocessors and microcontrollers contain on-board driver circuits that require the connection of an external crystal, and usually the dual capacitors as well. When an IC does not contain an integrated oscillator, discrete crystal oscillators are the most common solution. A variety of companies, including CTS, ECS, and Ecliptek, manufacture off-the-shelf oscillators that include the crystal and driver circuit in a single package. These components typically have four terminals: power, ground, clock, and an optional clock enable.

Quartz crystals are ubiquitous, because they are inexpensive and provide a relatively high degree of frequency stability over time and temperature. Frequency tolerance between 50 and 100 parts per million (ppm), or better than 0.01 percent, is easily obtained. In contrast, most ceramic crystals, usually called *ceramic resonators,* are less expensive and have tolerances an order of magnitude worse



than common quartz crystals. Ceramic resonators are used in very low-cost applications wherein accuracy is forsaken for small cost savings. Most digital systems use quartz crystals that cost approximately $1.00, because they are reliable, they provide an accurate time base, and the crystal's cost is a small fraction of the overall system cost.

Some applications require tighter tolerances than normal crystal oscillators provide. More precise manufacturing techniques and control over materials can yield tolerances of approximately 1 ppm. Below this level, temperature control becomes a significant factor in frequency stability. So-called *oven-controlled oscillators* are specially designed to maintain the crystal at a stable temperature to greatly reduce temperature as a variable in the crystal's resonant frequency. Using this technique, oscillators are available with tolerances on the order of one part per billion! Conner-Winfield and Vectron International, among others, manufacture these high-accuracy oscillator products.

Most digital systems do not require clock accuracy better than 50 or 100 ppm. However, jitter is another clock stability characteristic of concern. Each oscillator circuit is subject to a certain amount of jitter based on the tolerance of its components. Aside from an oscillator's inherent jitter specification, ambient noise can couple into the oscillator and cause additional jitter. Therefore, it is desirable to attenuate ambient noise on the power supply that might otherwise couple into the oscillator circuit. It is common to find various types of LC filters on the power leads of crystal oscillators. A basic pi-type topology is shown in Fig. 16.2, consisting of a ferrite bead with capacitors on each side to attenuate high frequencies with small capacitors and provide lower frequency response with a larger capacitor. This circuit attenuates differential noise and provides the oscillator with a cleaner power supply relative to its ground reference.

16.2 LOW-SKEW CLOCK BUFFERS

Once a stable clock source has been established, the signal must be distributed to all components that operate on that clock. Clock distribution is critical to a digital system, because synchronous timing analysis assumes the presence of a reliable and consistent clock. Conventional synchronous buses running between ICs require a common clock so that they can work together with a known timing relationship. A single bit on a synchronous bus essentially consists of an output flop on one IC that drives an external wire, possibly passes through some combinatorial logic, and then is sampled at the input of a flop on another IC. Each IC on the bus should ideally see the same clock signal. In reality, there are slight skew variations between these individual clocks. Some clocks arrive a little sooner than others. Skew should be kept to a minimum because, like jitter, it reduces the synchronous timing budget.

Common clock signals are distributed in a low-skew manner by closely matching the delays from the clock source to all loads. Distribution delays are incurred as the clock passes through passive wires and active buffers. Consider the hypothetical clock distribution tree shown in Fig. 16.3. An oscillator drives a clock buffer, which drives five loads. All of the clock signals are point-to-point with

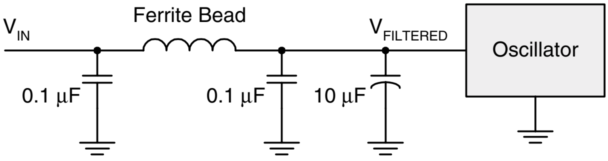


FIGURE 16.2 Oscillator LC pi power filter.

ripheral IC that is located close to it, the peripheral's clock input can be driven directly by the microprocessor's clock output, and all should be okay. Chances are great, however, that the microprocessor must communicate with several peripheral and memory ICs that are located a significant distance away, as shown in Fig. 16.7. Placing a low-skew buffer between the microprocessor's clock output and the other ICs will not solve the problem, because the clocks at the loads will have a large skew relative to the microprocessor clock.

Another common clock distribution problem with an absolute delay constraint is driving common clocks to an expansion board. Consider the system shown in Fig. 16.8 in which a base board con-

of the loop stability problems have been taken care of on a single IC, allowing engineers to use an integrated PLL as a tool and focus on the task of zero-delay clock distribution.

A zero-delay clock distribution circuit contains a PLL whose feedback path is chosen such that there is a nearly perfect phase alignment between the PLL reference clock and the clock loads. Figure 16.10 shows a general zero-delay clock distribution scenario. The feedback path has a low-skew relationship with the other clock loads. It is driven by the same low-skew buffer and is purposely routed on a wire that is length matched with the other clock loads. The result is that the feedback clock arrives back at the PLL at the same time that the other clocks reach their loads. Of course, a low-skew buffer has finite skew, and this skew limits the PLL's ability to align the output clocks to the reference clock. PLL feedback clocks are typically carried on wires that have been artificially lengthened by serpentine routing, because the PLL and buffer are usually close together while the clock loads are further away. It is actually the norm for a digital PLL to be integrated onto the same IC as a low-skew buffer, because the two functions are inextricably linked in digital clock distribution applications. Companies that manufacture low-skew buffers commonly offer integrated PLLs as well, often under the term *zero-delay buffer*.

Turning back to the microprocessor clock distribution example in Fig. 16.7, it is now apparent that a PLL with an integrated low-skew buffer would solve the problem. The microprocessor clock would drive the PLL reference, and matched-length wires emanating from the buffer would go to each clock load as well as the PLL feedback input. The resultant skew between the microprocessor's bus mawO88—micnAC©b©Hfw fmiG—Aen x—8ftrop-op—x—8fmax—8fwO88—x—8fm …p— —nUc the expansion board between the connector and the zero-delay buffer. It is easier to constrain a short wire length leading to the zero-delay buffer, because these ICs are typically small and can be located near the connector, regardless of the board's overall layout.

The downside of PLLs is that they are sensitive to jitter and may actually add jitter to an otherwise clean clock. A typical PLL is only as good as the reference clock that is supplied to it, because the phase detector seeks to continually adjust the VCO to match the incoming reference. As such, a typical PLL will not reduce jitter. A loop filter with a long time constant may be used to reduce jitter, although other problems can result from doing so. Noisy power increases a PLL's jitter, because the analog VCO circuit translates noise into varying oscillation periods. Therefore, PLLs should have their analog power supply filtered with at least a series impedance and shunt capacitance as shown earlier for a crystal oscillator using an LC pi filter.

16.4 FREQUENCY SYNTHESIS

Digital systems can use PLLs for more than just zero-delay clock buffering. Arbitrary frequencies can be synthesized by a PLL based on a reference clock, and this arbitrary frequency can be changed in real time. Systems with analog front ends and digital processing cores use frequency synthesis for tuning radios and implementing complex modulation schemes. A common example of this is a digital cellular telephone. Purely digital systems make extensive use of frequency synthesis as well. Advanced microprocessors and other logic ICs often run their cores at a multiple of the external bus frequency. PLLs are used as clock multipliers in these ICs.

Frequency synthesis is possible with a PLL, because the phase detector and VCO do not have to operate at the same frequency. The phase detector cares only that its two inputs are phase aligned; they must be of the same frequency and phase for a neutral error signal to be generated. Subject to minimum and maximum operating frequency limitations, the VCO can actually be made to run at any frequency. Let's first consider the example of synthesizing an integer divisor of the reference clock. The PLL circuit in Fig. 16.12 places a divide-by-N counter between the true reference clock and the phase detector's reference input. The result is that the PLL is unaware of the original reference clock and instead sees the divided version, whose frequency it is able to match by adjusting the VCO until the phase error is zero.

Clock division by an integer divisor is not very interesting, because it can be done without the complexity of a PLL. Now let's look at integer multiplication, which does require a PLL. The circuit in Fig. 16.13 places a divide-by-M counter in the feedback path between the VCO output and the phase detector. Things start to get interesting here. Instead of observing the true VCO signal, the phase detector unknowingly gets a divided version of that clock. The phase detector has no knowledge of what signals it is seeing—just whether they are phase aligned. Therefore, the phase detector emits an error signal until the divided feedback clock matches the frequency and phase of the reference. The VCO must run at a multiple of M times the reference for the feedback clock to equal the reference, resulting in clock multiplication.

Integer multipliers and dividers are useful, but truly arbitrary ratios between input and output can be achieved when the reference and feedback dividers are joined into a single circuit. The circuit in Fig. 16.14 allows the output to run at a ratio of M ÷ N times the reference frequency by combining the principles of the aforementioned divider and multiplier schemes. Different frequencies can be synthesized in real time by periodically changing the M and N counters. A PLL has a finite lock time, the time that the loop takes to adjust to a new operating frequency. Any frequency synthesis scheme that requires real-time adjustment must take the lock time into account. If the counters are changed too rapidly, the PLL may spend all of its time hunting for the new frequency and never settling.

PLLs in general are capable of frequency synthesis when complemented with external counters, but most digital PLLs are not designed specifically for large multiplication ratios. Several factors influence the suitability of a PLL for arbitrary clock synthesis, including VCO stability and loop-filter characteristics. VCO stability is improved by filtering its power supply. Typical core clock synthesis applications in a microprocessor or logic IC may require multiplication by 2 or 4, which is relatively straightforward. As the multiplication ratio increases, the control path between the phase detector and the VCO is degraded, because fewer feedback adjustment opportunities exist for every VCO clock period. If a PLL is multiplying a reference by 100, the phase detector will only be able to judge phase errors on 1 percent of the VCO edges. There is the potential for VCO wander during the long intervals between phase detector corrections.

Most digital PLLs contain a lowpass loop filter integrated on chip that is matched to the typical operating scenario of a 1:1 input/output ratio or some small multiplication factor. Larger multiplication ratios may require a more complex off-chip filter that is specifically designed for the application's requirements. This is where PLL design in a digital system can start to get complicated. Fortunately, semiconductor manufacturers have documentation and applications engineering staff available to assist with such tasks. One set of devices that are well suited for digital frequency synthesis is Texas Instruments' TLC29xx family. The TLC29xx devices have separately connected phase detector and VCO sections such that an external loop-filter can be customized to the application.

It is not very common for a digital system to require complex multiplication ratios in a PLL, but the requirement does exist in applications that have multiple interfaces that run at different frequencies and must be phase locked to prevent data loss. One example is a digital video processor. The primary clock frequency of many digital video standards is 27 MHz; however, some newer high-definition video standards operate at 74.25 MHz. These frequencies are related by a factor of 2.75. If a digital video processor must perform some conversion between these two interfaces and do so in a manner that precisely matches their data rates, a PLL is necessary to lock one of the interfaces to the other. Otherwise, each interface would run on its own oscillator, and small amounts of frequency drift would soon cause one to get a little faster or slower than the other. No matter how accurate an oscillator is, it has a finite deviation from its nominal frequency. When two oscillators are paired with the expectation that they run at constant rates, a problem will eventually develop. As soon as a rate mismatch develops, the input and output data rates no longer match, and data is corrupted.

For the sake of discussion, assume that the 27-MHz interface is the master, and the 74.25-MHz interface is the slave. Because a counter cannot directly divide by 2.75, a ratio of integers must be calculated. The smallest pair of integers that yields $M \div N = 2.75$ is $M = 11$ and $N = 4$. This means that the PLL is essentially performing an 11× clock multiplication function, because the 27 MHz reference is divided by 4 to yield a 6.75-MHz PLL input.

The task of designing a PLL to implement large multiplication factors is decidedly non-trivial because of the problems of stability and jitter. Selecting an appropriate lowpass filter that keeps the loop from unstable oscillations and that adequately addresses VCO jitter can involve significant control systems theory and analog filter design skills.

16.5 DELAY-LOCKED LOOPS

PLLs have traditionally been considered the standard mechanism for implementing zero-delay buffering and clock multiplication. Their flexibility comes at a certain price for manufacturers of digital ICs, because PLLs are analog circuits that must be isolated from noisy digital switching power supplies for low-jitter operation. Although the problems of on-chip isolation have been addressed for a long time, the problem persists. Many large digital ICs are now manufactured with a purely digital

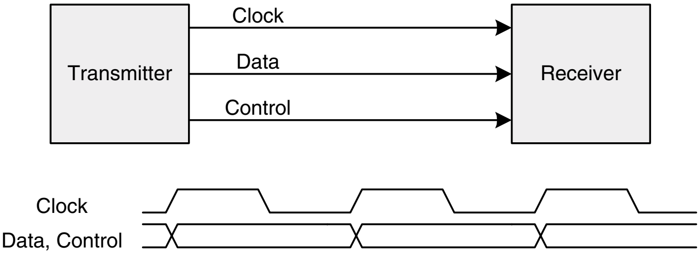


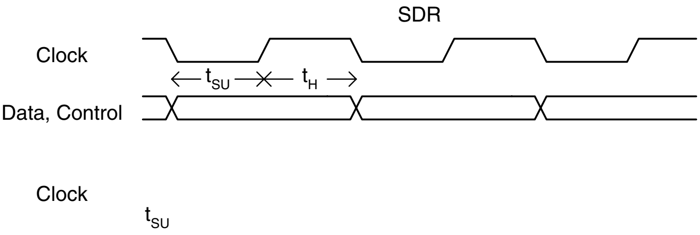
FIGURE 16.16 Source-synchronous bus.

uniform propagation delay. By the time the signals reach the receiving IC, clock and data are phase aligned within a certain skew error that is a function only of the transmitter's output-to-output skew and the delay mismatch of the PCB wiring. This skew is far less than the skew and wiring delay penalties of a conventional synchronous interface and may be on the order of 100 ps. Nearly the entire clock period, less allowances for finite skew and switching time, can be used to meet the input flops' setup and hold specifications, which is why high-performance memory (e.g., DDR SDRAM) and logic interfaces can run at hundreds of megahertz.

The clock and data timing relationship can be arbitrary as long as the skew is tightly controlled, because the receiver's input circuitry can implement appropriate delays based on the transition of data relative to clock to meet the flop's true setup and hold requirements. Two common source-synchronous timing relationships are shown in Fig. 16.17, where the clock is offset into the middle of the data valid window. In the case of single-data rate bus (SDR), this is akin to clocking off the falling edge. A double-data rate bus (DDR) would require that the clock be shifted by 90° relative to data such that the rising and falling edges appear in the middle of the valid windows so that data can be registered on both edges.

A disadvantage of source-synchronous bus architecture is the management of a receive clock domain that is out of phase with the core logic domain. A conventional synchronous design seeks to maintain a uniform clock phase across an entire design, whereas source-synchronous design explicitly gives up on this and proliferates disparate clock domains that have arbitrary phase relationships with one another. This trade-off in clock domain complexity is acceptable, because individual gates are cheap on multimillion-transistor logic ICs. A logic IC typically requires a FIFO and associated control logic to cross between the receive clock and internal clock domains.

Source-synchronous interfaces are often point-to-point, although this is not strictly necessary, because of the high speeds at which they usually run. It is electrically difficult to fabricate a multidrop



bus that operates at hundreds of megahertz. Figure 16.18 shows a hypothetical system architecture that employs source-synchronous interfaces. There are three ICs, and each IC is connected to the other two via separate interfaces. Each interface consists of two unidirectional buses. Unlike the receive clock, the transmit and core clock domains are merged, because this example does not require the core to operate at a different frequency from that of the bus. It would be easy to insert a FIFO, too, if a benefit would result from decoupling the bus and core frequencies. Clock distribution for this system is relatively easy, because each IC requires the same core clock frequency, but the interfaces are self-timed, which removes clock tree skew as a concern.

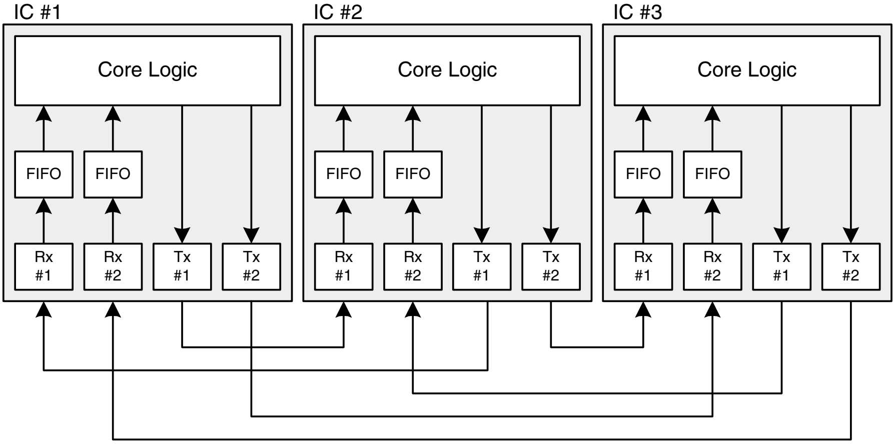


FIGURE 16.18 Source-synchronous system architecture.

17.1 VOLTAGE REGULATION BASICS

Power usually enters a system in a form dictated by its generation and distribution characteristics rather than in a form required by the components within that system. Plugging a computer into a 120 VAC wall outlet provides it with power in a form that was deemed efficient for generation and distribution in the early twentieth century. Unlike light bulbs, digital logic chips do not run very well if connected directly to 120 VAC, although they may emit a bright light for a brief period if connected in this manner! Rectifiers and the conversion between AC to DC has already been discussed, but this is only part of the solution to providing usable power to system components. Once the AC has been rectified to DC and filtered, the voltage probably does not meet the specifications of the circuit. A rectified power input may not only have the wrong DC level, it probably has a good deal of ripple artifacts from the original AC input, as shown in Fig. 17.1.

An ideal voltage regulator provides a constant DC output without ripple regardless of the input voltage's DC level and ripple. This applies not only to rectified AC power but also to power provided by batteries, solar cells, DC generators, and so on. Any time a system's power input does not provide the supply voltage required by its components, a voltage regulator is necessary to perform this conversion. Most digital systems require at least one voltage regulator, because it is rare to find a power source that provides the exact voltage required by digital and analog circuits. There are, however, some special-purpose ICs designed with wide supply voltage specifications so that they can be directly connected to batteries without an intermediate voltage regulator. On the flip side, there are systems that require multiple voltage regulators, because they contain circuitry with multiple supply voltage specifications. At one time, it was common to have a single +5-V digital supply. Now, it is not uncommon to have 3.3-, 2.5-, 1.8-, and 1.5-V supplies in myriad configurations.

Most voltage regulators are the *step-down* variety—they provide a constant output voltage that is lower than the input. There are many types of step-down regulators, as will be shown in the course of this chapter. Some applications require *step-up* regulators that provide a constant output that is higher than the input. These applications are often low-power battery-operated devices in which a 1.5- to 3-V battery is moderately stepped up to power a small circuit.

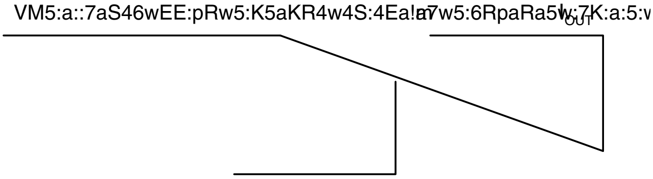
A voltage regulator must have access to some form of voltage reference to which it can compare

than half the rated $T_{J(MAX)}$ of 150°C. These estimates point in the right direction. However, the best approach is to confirm these findings with the manufacturer. Semiconductor manufacturers' application engineers are able to answer such questions and provide advice on matters that are not explicitly addressed in a data sheet.

As with the Zener shunt regulator, this circuit is relatively loose in its accuracy of V_{OUT} over varying temperature and load. The Zener voltage reference itself will drift with temperature, but its current is nearly static, so there will not be much drift with changing current. The transistor's V_{BE} changes with temperature and current as well. If these drawbacks have you wondering how accurate voltage regulators are ever constructed, the answer lies in various compensation schemes that involve more components. However, the biggest contributor to accurate voltage references and regulators is the integrated circuit, because an IC enables the pairing of transistors with closely matched physical and thermal characteristics. When transistors and diodes are fabricated on the same slice of silicon within microns of each other, they are nearly identical, and they operate at the same temperature. Close matching enables transistors and diodes to largely cancel out each other's undesired variations when arranged in specific configurations.

A series regulator can also be designed to provide a regulated constant current instead of constant voltage. Current sources are useful for battery chargers, among other applications. Figure 17.8 shows such a circuit using a PNP transistor. Once again, the variable input/output differential is taken up by the transistor's V_{CE}, although it is the output that is allowed to float with a constant current. A floating output voltage is necessary, because an ideal current source supplies constant current regardless of the impedance that it is driving. A higher load impedance results in a higher voltage output according to Ohm's law. This is the converse of a voltage source wherein constant voltage is desired at variable current. Real current regulators, of course, have limitations on the range of V_{OUT} for the circuit to remain in regulation, just as we have already observed that voltage regulators are subject to current limitations.

The TIP32 is chosen for this circuit, because it is a mature BJT with characteristics similar to the TIP31. Power circuits that require both NPN and PNP transistors sometimes use the complementary TIP31 and TIP32 pair. This current regulator functions by establishing a fixed voltage drop across R_{SET}, thereby establishing a fixed emitter current. Assuming negligible base current, the collector current drives the load with the same current. An emitter-base loop is established with a reference voltage provided by the 1N4728A Zener diode. Per loop analysis, $V_{ZENER} = I_{OUT}R_{SET} + V_{BE}$, assuming that the base current is negligible. When the Zener reference and V_{BE} are fixed, R_{SET} establishes the regulator's output current. R_{LIMIT} picks up the voltage difference between V_{IN} and V_{ZENER} and thereby serves as the Zener diode's current limiter.



The current regulator cannot supply a constant 100 mA for all values of V_{OUT}. As the load impedance is increased, there is a corresponding increase in V_{OUT}, which reduces the difference with respect to V_{IN}. This differential eventually forces the transistor into saturation when its collector-base junction becomes forward biased. Using the estimate of $V_{CE(SAT)}$ = 0.3 V provides us with a minimum voltage drop across the transistor for the regulator to maintain 100 mA. The voltage drop across R_{SET}, 2.6 V, must also be factored into the V_{IN}/V_{OUT} differential. The maximum V_{OUT} must be evaluated along with minimum V_{IN} to analyze the limiting scenario: $V_{OUT(MAX)} = V_{IN(MIN)} - 2.6\text{ V} - 0.3\text{ V} = 7.1\text{ V}$. This circuit can therefore be guaranteed to function properly up to 7.1 V. Of course, if V_{IN} is at the upper end of its range, V_{OUT} can be higher as well. But the worst-case scenario is what should be used when specifying the guaranteed parameters of a circuit.

Analysis of the regulator's dropout voltage shows that the drop across R_{SET} is the dominant term within the regulator circuit. If dropout voltage is a concern, the drop across the resistor should be minimized. This is why a low-voltage Zener has been selected rather than a 5- or 6-V diode. The drop across R_{SET} is the difference between V_{ZENER} and V_{BE}. Minimizing V_{ZENER} minimizes the drop across R_{SET}.

Calculating power dissipation to perform a thermal analysis requires bounding the output voltage at a practical minimum, because the transistor's power dissipation increases with increasing V_{CE} and, hence, decreasing V_{OUT}. R_{SET} has a constant voltage drop at constant current, so it has constant power dissipation of 0.26 W. For the sake of discussion, we can pick $V_{OUT(MIN)}$ = 0 V. Along with $V_{IN(MAX)}$ = 15 V, the power dissipation of the transistor is $I_{OUT}(V_{IN} - I_{OUT}R_{SET} - V_{OUT})$ = 1.24 W. The TIP32 and TIP 31 have equivalent power and thermal ratings. The transistor is rated at 2 W without a heat sink at an ambient temperature of 25°C, and a conservative design methodology might call for a heat sink on the TO-220 package in this case. A heat sink enables calculation of the package temperature for a given power dissipation and ambient temperature, which in turn enables us to take advantage of the manufacturer's power derating curve expressed in terms of package temperature.

A small TO-220 heat sink can provide a thermal resistance of 30°C/W with natural convection. Much lower values are achievable when a fan is blowing air across the heat sink and with a larger heat sink. A temperature rise of 37°C is attained with P_D = 1.24 W. Therefore, a 40°C ambient temperature would result in a transistor case temperature of 77°C. The TIP32's power derating curve shows that the transistor is capable of over 20 W at this case temperature.* The heat sink thereby enables a very conservative design with minimal cost or complexity.

17.5 LINEAR REGULATORS

Most voltage and current regulation requirements, especially in digital systems, can be solved with ease by using integrated off-the-shelf regulators that provide high-quality regulation characteristics. Constructing a regulator from discrete parts can be useful when its requirements are sufficiently outside the mainstream to dictate a custom approach. However, designing a custom regulator brings with it the challenges of meeting the load's regulation requirements over a potentially wide range of operating conditions. Power supplies for digital circuits have the benefit that the voltages are common across the industry. This has enabled semiconductor manufacturers to design broad families of integrated regulators that are preadjusted for common supply voltages: 5, 3.3, 2.5, 1.8, and 1.5 V. Manufacturers also offer adjustable regulators that can be readily customized to a specific output voltage. The result is the ability to treat regulators largely as "black boxes" once their overall charac-

* TIP32 Series, Fairchild Semiconductor, 2000, p. 2.

the current through R1 is more than 6 mA—more than two orders of magnitude larger than I_{ADJ}. The actual error imparted by I_{ADJ} = 100 μA is I_{ADJ}R2 = 41.2 mV, or roughly 1 percent of the 3.3-V output. This error can be reduced by increasing the programming current, which causes a corresponding decrease in R2.

The LM317 has two key requirements to maintain a regulated output. First, a minimum load current of 10 mA is necessary. This is partially addressed via the programming current—6 mA in our example. If the load cannot be guaranteed to sink the remaining 4 mA, a 120-Ω resistor can be substituted for R1 with an accompanying adjustment to R2. Second, the input must exceed the output by the LM317's dropout voltage, which is as high as 2.5 V, depending on load current and temperature. Dropout voltage decreases with decreasing load current and varies nonmonotonically with temperature.

A further improvement to the basic LM317 circuit is to add a 10 μF bypass capacitor between the adjustment pin and ground to filter ripple noise being fed back through R1. This improves the regulator's ripple rejection. When this capacitor is added, the same issue of safe power-off comes up, because the capacitor will hold charge and discharge through the adjustment pin. As before, the solution is a diode with its anode connected to the adjustment pin and its cathode connected to the output pin. This provides a low-impedance path from the bypass capacitor through two diodes to the input node.

27-Ω dropping resistor is inserted before the regulator. The dropping resistor is sized based on the load current, the minimum input voltage, and the regulator's dropout voltage. Working with a conservative dropout voltage of 3 V and a minimum input level of 21.6 V, the resistor must drop a maximum of 13.6 V at 500 mA. The closest standard resistor value, 27 Ω, yields a drop of 13.5 V at maximum load with a power dissipation of 6.75 W. Finding a power resistor that can safely dissi-

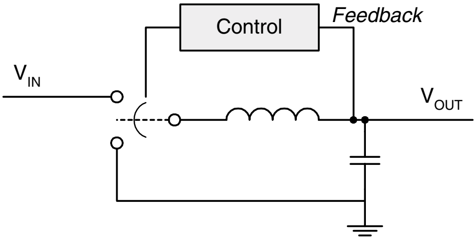


FIGURE 17.13 Conceptual step-down regulator.

capacitor opposes voltage changes by sinking or sourcing current, the inductor changes its voltage to maintain a constant current flow. When the switch selects ground, the inductor instantly flips its voltage so that it can continue supplying current to the capacitor. The inductor holds a finite amount of energy that must be quickly replenished by switching back to the input voltage. The role of the feedback and control circuit is to continuously modify the switching frequency and/or duty-cycle to maintain a fixed output voltage.

There are varying designs for switching circuits. Two common topologies often seen today are shown in Fig. 17.14. Most modern switching regulators employ power MOSFETs because of their low R_{DS}. Prior to the availability of power FETs, power BJTs were used as switches, and their finite $V_{CE(SAT)}$ resulted in higher losses than seen with modern FETs. When a single transistor is used as the switch, a diode serves as the ground shorting element, or rectifier. The diode is reverse biased when the transistor is conducting, and it becomes forward biased after the switching event causes the inductor's voltage to flip. The inductor changes its voltage to maintain a constant current flow, which causes its switch-side voltage to suddenly drop, and the diode clamps this dropping voltage to near ground. Substantial current flows through the diode and motivates the selection of a low forward-voltage Schottky diode (note the S-curve symbology for a Schottky diode). Power loss in this diode is a major source of switching regulator inefficiency. This has given rise to the dual-transistor switch circuit that replaces the diode with a FET as the main rectification element. When a transistor is used in this manner as a rectifier, the common industry term is *synchronous rectification*. The FET's low R_{DS} makes it a superior solution to the fixed voltage drop of the Schottky diode. However, a diode is still present to serve as a rectifier during the short but finite turn-on time of the bottom FET.

Minimal power loss in the switching transistors is a key attribute that enables high-efficiency switching regulators. A switch transistor is ideally either on or off. When off, the transistor dissipates no power. When on, there is minimum voltage drop across the transistor and, hence, minimal power

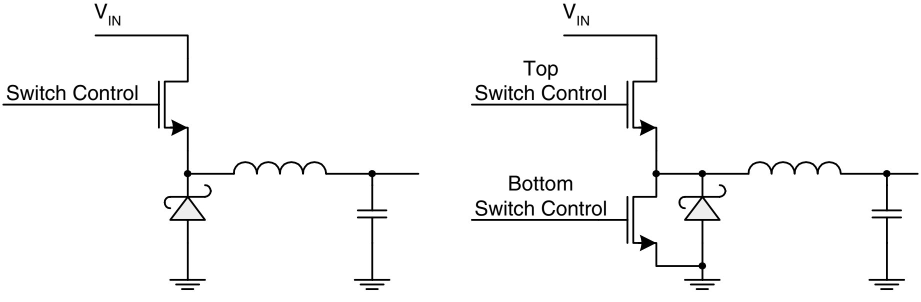


FIGURE 17.14 Single and dual FET switching circuits.

entities for a couple of reasons. First, they are usually associated with different levels of safety standards. AC power supplies must conform to strict standards, because they have the potential to short circuit an AC wiring system with the resultant risk of serious injury and damage. Second, DC-to-DC converters are tailored specifically to the voltages required by a digital logic circuit, whereas AC power supplies can provide more standardized voltages given the widespread availability of DC regulators.

The example also shows two distinct ground nodes: Earth ground and signal ground. A grounded AC wall outlet or other connection provides an absolute 0-V Earth ground connection into which excess charge can drain. This path prevents the accumulation of charge to the point of damaging sensitive electronic components when a sudden electrostatic discharge may occur. It also ties separate pieces of equipment to the same ground potential so that they can be connected without adverse consequences. For example, if a printer and a computer are both connected via a cable, and each has a different ground potential, a ground loop may develop whereby unexpected current flows between the two dissimilar voltages. A ground loop can be disruptive to communication between the printer and computer by manifesting itself as noise on the cable. In contrast to Earth ground, signal ground is the return path to the voltage regulators. Signal and Earth ground are usually connected so that there is a uniform DC ground potential throughout a system. However, many styles exist for making this connection. Some engineers and situations favor connecting the two grounds at a single point, often in the AC power supply. Others favor connecting the grounds at many points throughout the overload protection mechanisms that shut down in the event of excessive current draw. Safety standards for DC-to-DC converters are less uniform and strict, because it is often sufficient to rely on the safety cut-off mechanisms in the AC power module. If a DC regulator fails, the worst-case scenario is often that the AC supply's overload protection will be activated and prevent damage to other equipment and the AC wiring infrastructure.

Whenever power is distributed between two points, the conductors carrying that current must be adequately sized for the flow. Wires have current ratings based on their resistance, the ambient operating temperature, the maximum allowable insulation temperature, and the number of wires bundled together. It is important to conservatively specify wire capacity, because wire has a positive temperature coefficient, meaning that resistance increases with temperature. If a wire is operated beyond its safe capacity, a dangerous situation can develop in which heating increases resistance, which causes more heating in a self-destructive cycle. Wire manufacturers should always be consulted on the ratings for their products when selecting the necessary gauge wires for an application. The American Wire Gauge (AWG) standard provides a measure of wire size, with thicker wires indicated by smaller gauge numbers. Table 17.2 lists the resistance of solid copper wire per 1000 ft (304.8 m) at 25°C.* Current ratings are based on the allowable temperature rise over ambient, which is why a wire's environment directly affects its rating. AC wires enclosed in the walls of your home are rated more conservatively because of the risk of fire in confined spaces. The conservative use of lower-gauge wire results in less power loss and heating, with a resulting increase in safety and reliability.

TABLE 17.2 Resistance of Solid Copper Wire at 25°C

Wire Gauge (AWG)	Ω per 1000 ft (304.8 m)	Wire Gauge (AWG)	Ω per 1000 ft (304.8 m)
30	104	20	10.0
28	65	18	6.4
26	41	16	4.0
24	26	14	2.5
22	16	12	1.6

An example of a distributed power regulation scheme is shown in Fig. 17.16. This system uses an off-the-shelf AC-to-DC power supply to provide a 12-VDC intermediate power bus. Common intermediate voltages include 48, 24, 12, and 5 V. The advantage of using a higher voltage is less current flow through the intermediate distribution wiring for a given power level and hence lower resistive losses ($P_D = I^2R$) in that wiring. Lower voltages have the benefit of easier component selection because of the lower voltage ratings. When using a switching regulator, there is a compromise between very low and very high input voltages. Too high an input requires a small switching duty cycle, which results in higher losses as the transistors turn on and off more often relative to the time that they are in a static state. Too low an input causes higher current to be drawn from the source, which leads to higher I^2R losses in the regulator components. Often, 12 V is a good compromise between switching losses and easier regulator design. Additionally, some systems require 12 V for analog interface circuits or low-power motors such as a disk drive. This system can use whatever 12-V power supply is easily available, as long as its capacity is greater than 43 W.

* *The ARRL Handbook for Radio Amateurs,* American Radio Relay League, 1994, pp. 35–36.

leaded capacitors. Whereas a leaded component can have inductance of many nanohenries, Table 17.2 lists approximate inductances near 1 nH for various sizes of 0.1-μF surface mount ceramic capacitors.* Capacitors, resistors, and inductors are available in standard sized surface mount packages that are designated by their approximate dimensions in mils. Common sizes are 1210, 1206, 0805, and 0603. Larger packages such as the 1810 exist for handling higher power levels. Smaller packages such as the 0402 are used when space is at an absolute premium, but handling such packages whose dimensions are comparable to grains of sand requires special equipment.

TABLE 17.3 Surface Mount Capacitor Lead Inductance

Package	Inductance (pH)
1210	980
1206	1,200
0805	1,050
0603	870

The impedance of a 0.1-μF surface mount capacitor reaches a minimum value of under 100 mΩ at around 10 MHz and remains below 1 Ω from approximately 1 MHz to 100 MHz. This explains why 0.1-μF capacitors have been popular as bypass capacitors for so long: many digital systems have switching frequencies below 100 MHz. Above 100 MHz, 0.01-μF capacitors in 0603 packages become attractive because of their lower impedance at higher frequencies. It gets harder to reduce very high-frequency noise, because the inductance of surface mount bypass capacitors declines only to a certain point.

Having discussed the basic issues of power distribution, attention can be turned back to the example in Fig. 17.16. Distributing the regulated 5-, 3.3-, and 2.5-V logic-level supplies in a manner that ensures high-frequency electrical integrity requires minimal impedance between the regulator and the load. The consideration goes beyond DC resistance to include inductance, which has a more substantial impact on the conductor's impedance at high frequency. As the regulator and load are separated by higher impedance, the regulator's ability to respond to fluctuations in load current is degraded. Low-inductance distribution is achieved by using complete planes or very wide copper paths to connect the regulator outputs to the various ICs and components that they serve.

Power plane design is directly related to the electrical integrity of other signals in the system and will be covered in more detail later. The ideal situation is to devote entire PCB layers to serve as power planes for each separate voltage. This eliminates power plane cuts that can cause other signal integrity problems. Unfortunately, not all systems can afford the cost or, in some cases, the physical size of many power planes. In a situation like this, multiple voltages must share the same PCB layer. Figure 17.17 shows a hypothetical single power plane structure for the preceding example that requires distribution of 12, 5, 3.3, and 2.5 V. The shaded regions represent continuous copper areas. A ground return plane is required but not shown, because it occupies a second layer and is continuous across all power plane regions. Each system has its own unique power distribution flow governed by the grouping of components that require different supply voltages. This example assumes the com-

* Jeffrey Cain, *Parasitic Inductance of Multilayer Ceramic Capacitors*, AVX Corporation, p. 3.

This page intentionally left blank.

CHAPTER 18
Signal Integrity

Getting high-speed digital signals to function properly is one of those areas that many people call *black magic.* It is so called, because fast digital signals behave like the analog signals that they really are, which is not an apparent mode of operation from a binary perspective. The typical story of woe is one in which a digital engineer continues to design faster circuits in the same way as slower circuits, and one day a system begins developing unexplained glitches and problems. From a digital perspective, nothing substantial has changed. On closer inspection with an oscilloscope, individual digital signals have mysterious transients and noise superimposed on them.

Signal integrity is the overall term for high-speed electrical design techniques that enable digital signals to function digitally in the face of physical phenomena that would otherwise cause problems. Many of the terms and techniques introduced in this chapter may sound familiar, because they have received increased scrutiny and coverage in the trade press and at conferences as a result of the steady increase in semiconductor operating frequencies. Signal integrity used to be a topic that many systems could ignore simply by virtue of their older technology and slower signals. That luxury has largely evaporated today, even for slow systems that unwittingly use ICs designed for high-speed operation.

A broad set of topics are discussed in this chapter with the goal of providing familiarity with signal integrity problems and general solutions to those problems. Transmission lines and termination are absolutely critical interrelated subjects, because they literally make the difference between working and nonworking systems. Transmission lines address head-on the reality that wires have finite propagation delay and are not ideal transparent conductors that ferry signals from point to point unchanged. From a purely functional perspective, proper transmission line analysis and design is the most important part of signal integrity, which is why these topics are presented first.

High-speed signals exist in a world of non-negligible electromagnetic fields that cause even small wires to act as antennas. These antennas are capable of both radiation and reception of noise. Crosstalk, electromagnetic interference, and electromagnetic compatibility are associated topics that hinge around the reality that electrical signals do not remain neatly confined to the wires on which they travel. The problems are twofold. First, excessive field coupling can cause a circuit to malfunction. Second, electronic products offered for sale in most countries of the world must comply with government regulations regarding their electromagnetic emissions. You don’t want to bring home a new DVD player to find that it crashes your computer when you turn it on!

The chapter concludes with another related topic, electrostatic discharge. Static electricity is something that we are all familiar with, but its effects on a digital system are potentially disruptive and even destructive. Static electric discharges cannot be prevented in normal environments, but their effects can be reduced to the point of not causing problems.

18.1 TRANSMISSION LINES

Transmission lines, reflections, and impedance matching have been alluded to previously. The term *transmission line* can refer to any conductive path carrying a signal between two points, although its usual meaning is in the context of a conductive path whose length is significant relative to the signal's highest-frequency component. Circuits are normally drawn assuming ideal conductors whose lengths are negligible and assuming that the voltage at any instant in time is constant across the entire conductor. When a wire "becomes a transmission line," it means that it can no longer be considered ideal. An electrical signal propagates down a wire with finite velocity, which guarantees that a changing signal at one end will take a finite time to reach the other end. When a signal's rate of change is slow relative to the wire's delay, many nonideal characteristics can be ignored. Older digital circuits that ran at several megahertz with slow transition times were often not subject to transmission line effects, because the wire delay was short compared to the signal's rate of change.

A signal that changes rapidly forces one end of a transmission line to a significantly different voltage from other points along that conductor. At the instant this rapid change is produced by a driver, the signal has not yet reached the load at the end of the wire. Rather than observing current and voltage that are in proportion to the load impedance, they are in proportion to the characteristic impedance of the transmission line, commonly written as Z_O. Z_O is not a DC load; it represents the reactance developed by the conductors' inductive and capacitive characteristics. It is the impedance that would be observed between the two conductors of an infinitely long transmission line at nonzero frequency. When a high-frequency signal transitions before the driver sees the end load, it is as if the transmission line is infinitely long at that moment in time.

Transmission lines are composed of a signal path and a return path, each of which can be modeled using discrete lumped elements as shown in Fig. 18.1. The model shown is that of an unbalanced transmission line wherein all of the inductive and lossy properties are represented in one conductor. This is acceptable for many transmission lines in a digital system, because printed circuit boards commonly consist of etched wire conductors adjacent to ground planes that have negligible inductance and resistance. A balanced transmission line model, such as that representing a twisted pair cable, would show series inductance and resistance in both conductors. Analysis is simplified by assuming lossless conductors, which is often a suitable starting point in a digital system with moderate wire lengths. Using this simplification, the characteristic impedance is defined as $Z_O = \sqrt{L \div C}$.

Characteristic impedance is an important attribute, because it defines how a high-speed signal propagates down a transmission line. A signal's energy can fully transfer only between different transmission line segments that have equal Z_O. An impedance discontinuity results when two transmission lines are joined with differing Z_O. Impedance discontinuities result in some of a signal's energy being reflected back in the direction from which it arrived. This phenomenon is the crux of many signal integrity problems. An improperly terminated transmission line has the potential to cause reflections from each end of the wire so that the original signal is corrupted to the point of being rendered useless. A reflection coefficient, represented by the Greek letter gamma (Γ), that determines the fraction of the incident voltage that is reflected back from an impedance discontinuity is defined in the following equation:

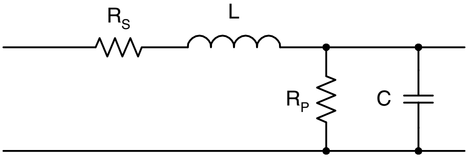


FIGURE 18.1 Lumped transmission line model.

cuits use 50-Ω transmission lines, and some connectors are available with this impedance as well. There are advantages and disadvantages in using higher- or lower-impedance transmission lines. A lower-impedance transmission line requires a thinner dielectric between the signal and return paths. This smaller gap makes the transmission line less susceptible to radiating and coupling noise. It also allows multilayer circuit boards to be made thinner. The disadvantage of lower-impedance transmission lines is that they require higher drive current. Most ICs are capable of driving 50-Ω transmission lines, and high-speed design makes the trade-off of improved noise immunity worth the added power consumption. However, slower circuits may be more suited to 75-Ω transmission lines for power savings.

18.2 TERMINATION

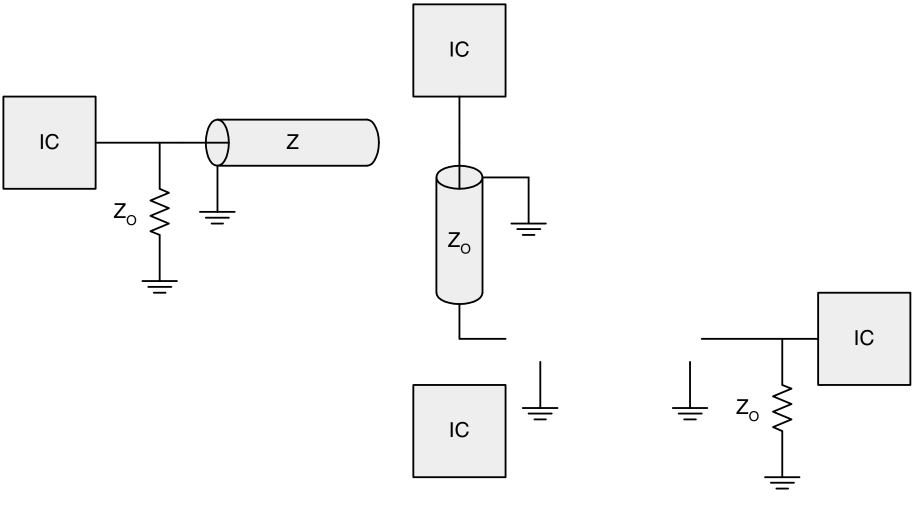
In applying transmission line theory to a digital signal, one should first determine whether the combination of signal transition speed and wire length combine to merit transmission line analysis. Digital signals are characterized not only by their repetitive frequency but also by the frequency components in their edges. A signal with a rapid transition time will have high-frequency components regardless of how long the repetition period. Rules of thumb vary, but a common reference point is to treat a wire as a transmission line if the signal's rise time is less than four times the prop-to accomplish the impossible but to achieve a transmission line topology in which the reflections are small enough to not degrade the load's valid detection of logic-1 and logic-0 states.

Parallel termination resistors should always be placed as close to the end of a transmission line as possible to minimize stub lengths between the terminator and the IC pin. Stubs appear as transmission lines of their own and can cause more reflections if not kept short. It can be difficult to squeeze termination components close to IC pins, but efforts should be made to achieve the best practical results.

Terminating a line at one end as shown in Fig. 18.6 is proper for a unidirectional signal, because the driver launches a signal into one end of the transmission line, and the termination is placed at the load end to prevent reflections. When both ends of the line are driven, as is the case with bidirectional buses, both ends require termination. The resistor at the driver end appears as a normal DC load, and the resistor at the far end serves as a terminator.

Situations commonly arise in which a bus has more than one load. A microprocessor bus must typically connect to several memory and peripheral ICs. The transmission line topology must be laid out carefully to minimize the potential for harmful reflections. The best scenario is to create a single, continuous transmission line terminated at each end that snakes through the circuit board and contacts each IC so that the stubs to each IC are of negligible length as shown in Fig. 18.7. When an IC at either end drives the bus, it drives a single transmission line that is terminated at the other end. When an IC in the middle drives the bus, it drives two equivalent transmission lines that are terminated at their ends. Graphically, the single transmission line can be drawn as shown using multiple segments connected by nodes that indicate tap points for individual ICs. Because nodes are drawn with the assumption of negligible length and constant voltage, they are conceptually transparent to the transmission line segments on each side. Keep in mind that not all buses require a perfect transmission line topology. Depending on their wire lengths and the switching times of the drivers, the wires may be regarded as idealized and not require special handling.

Many nonideal topologies exist in which no attempt has been made to shorten stubs and there is really no identifiable transmission line "backbone." Instead, the wiring is fairly random. A topology like this works either by virtue of the fact that the signal transition times are slow enough to not



The reflected signal propagates back to the driver. This time, the transmission line end is terminated by the resistor connected to a power rail via the driver circuit itself, and the reflected energy is absorbed. There is no DC power dissipation with series termination, because the terminating resistor does not shunt the transmission line to a separate DC potential.

The reflection intentionally created in a series-terminated transmission line makes this scheme nonideal for high-speed multidrop buses, because it takes two round-trip times for the entire transmission line to stabilize. A 12”-in (0.3 m) bus would require approximately 4 ns for the transmission line to settle, which is a substantial fraction of the timing budget at speeds over 100 MHz.

Bidirectional point-to-point transmission lines can use series termination as well, with good results. Figure 18.12 shows a transmission line with series termination at each end. The mode of operation is the same as explained previously. When component A is driving, R1 serves as the series termination, and the signal propagates toward R2. R2 connects to the high-impedance input circuit at component B, effectively nullifying the presence of that resistor. A reflection is developed at the R2 end of the transmission line and is absorbed when it returns to the R1. Some delay and lowpass filtering of the signal may result because of the RC time constant formed by R2 and any stray capacitance at component B’s input node. If the stray capacitance is up to 10 pF, the time constant is up to 500 ps—small, but non-negligible for very high-speed circuits.

Selecting the perfect series termination resistor is an elusive task, because it is difficult to characterize a driver circuit’s actual output impedance. This finite impedance combines with the series resistor to yield the total termination impedance seen by the signal reflecting back from the load. A driver circuit’s output impedance varies significantly with temperature, part-to-part variation, supply voltage, and the logic state that it is driving. It is therefore unrealistic to expect perfect series termination across time and multiple units manufactured. Some devices that are specifically designed for point-to-point transmission line topologies (e.g., certain low-skew clock buffers) contain internal series termination circuits that are designed to complement the driver’s output impedance. In the remaining cases, standard resistance values are chosen with the understanding that an imperfect termination will result. A typical value of 39, 43, or 47 Ω can be chosen for an initial prototype build when using 50-Ω transmission lines, and the signal integrity can be evaluated in the laboratorT©GuppOrna©fintexo8D88©HC*°C8DUU8O…_9Of thHGo÷fIn*re-x[wn be ch9A©fthe x9—G—fspE8©© an©

and so 8.7 percent of the reflected half-amplitude signal will be re-reflected. This may not be a sufficient amplitude disturbance to cause problems. If it is, the transmission line must be given time for the reflections to diminish.

18.3 CROSSTALK

Initial transmission line analysis is typically performed with assumptions of ideal circumstances, including the assumption that the transmission line is independent of others. In reality, a wire acts as an antenna and is a radiator and receiver of electromagnetic fields. When two nearby wires couple energy between each other, the phenomenon is called *crosstalk* and is another source of signal integrity problems. Crosstalk is not always a problem, but the potential exists, and therefore circuit design and layout should be performed with its consideration in mind.

Energy can be coupled between nearby conductors either capacitively or inductively. High-frequency energy can pass through a capacitor, and a small capacitor is formed when two conductors are in proximity to one another. The capacitance between two wires is a function of their surface area and their spacing. When two wires are run parallel to one another on the same layer of a printed circuit board, their mutually facing surface area is relatively small. A dual stripline configuration, however, can present greater capacitive coupling problems, because wire traces may run parallel one on top of the other with significant surface area. A common PCB routing rule is to route adjacent dual stripline layers orthogonally whenever possible rather than parallel to each other as shown in Fig. 18.13. Minimizing the surface area of a wire that is in close proximity to the other wire reduces capacitive coupling.

Inductive coupling comes about because current flowing through a wire generates a magnetic field. Each wire is a very small inductor. If two wires are run close to each other, the two small inductors can couple their magnetic fields from one to the other. Crosstalk analysis uses the terms *aggressor* and *victim* to aid in analysis. The aggressor is a wire that has current flowing through it and is radiating an electromagnetic field. The victim is a nearby wire onto which the electromagnetic field couples unwanted energy. Because the intensity of the magnetic field is proportional to the current flowing through a wire, heavier loads will result in more coupling between an aggressor and nearby victims. Most crosstalk problems in a digital system are the result of magnetic fields, because of the high currents resulting from low-impedance drivers and fast edge rates.

Separation is an effective defense against crosstalk, because electromagnetic field coupling decreases with the square of distance. Doubling the separation between two wires reduces the coupling at the victim by 75 percent. Dielectric height in a PCB is another contributing factor, because the field intensity increases with the square of height between the aggressor trace and the ground plane. The dielectric ranges in thickness according to the desired characteristic impedance and width of the

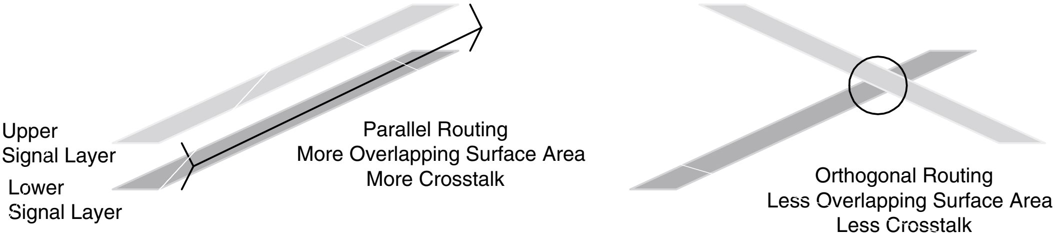
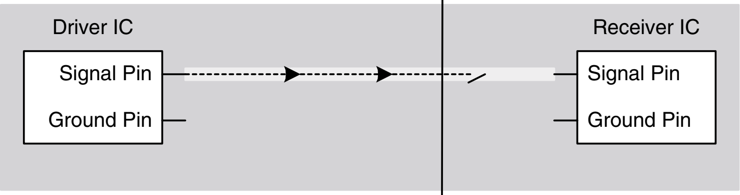


FIGURE 18.13 Dual stripline coupling reduction.



Capacitive coupling can occur through bypass capacitors that connect the two return planes and through interplane capacitance that is a function of the PCB dimensions and materials. Low-inductance capacitors are critical to provide adequate high-frequency coupling between planes. Impedance is a product of inductance and frequency. A bypass capacitor's total circuit inductance is the sum of the capacitor's inherent inductance and additional inductance caused by PCB traces and vias that connect it to the return planes. Typical surface mount capacitors in 0603 or similar packages can exhibit total inductance of approximately 2 to 3 nH. This inductance, combined with typical capacitance values of 0.1 or 0.01 μF, allow an impedance calculation at a given frequency. When multiple bypass capacitors are in close proximity to a via, they form a parallel combination with lower total inductance and higher total capacitance—both of which are desirable characteristics. The closer the bypass capacitors are to a via in question, the smaller the loop that is created for high-frequency return current between two planes. Nearby capacitors improve the high-frequency characteristics of the transmission line, whereas more distant capacitors increase the circuit's EMI susceptibility.

Discrete bypass capacitors are often the path of least impedance between return planes. As operating frequencies rise, however, the finite inductance of discrete components becomes more of a problem. A PCB may be constructed with planes separated by very thin dielectrics to provide significant interplane capacitance with negligible inductance. Capacitance increases with decreasing spacing between planes, and inductance decreases with greater surface area that a plane offers. High-frequency systems may require such construction techniques to function properly. These techniques can increase system cost by requiring more expensive thin dielectric materials and a greater number of PCB layers. Such costs are among many complexities involved in creating high-performance systems and must be considered when deciding on the practicality of a design.

The potential for via-induced EMI problems always exists. Conservative designs attempt to route sensitive and very high-speed signals with a minimum number of vias to reduce ground discontinuity problems. When vias are necessary, it is best to switch only between pairs of signal layers that are on opposite sides of the same ground plane.

Return path discontinuities may also be caused by breaks in power and ground planes. A PCB may contain multiple DC voltages (e.g., 5 and 3.3 V) on the same power plane to save money. Different voltage regions are created by splitting the plane. Return path discontinuities may result if the split plane is used as an AC ground for adjacent signal layers as shown in Fig. 18.16. As with a via, the return current finds the path of least impedance. An isolated split plane will force the return current to find an alternate path. Ideally, this should not be done, and planes should be continuous. Most engineers strive to never route a trace across a plane split so that potential signal integrity problems are minimized. If a plane does need to be split, it can be isolated from adjacent signal layers with additional solid ground planes. Alternatively, an adjacent plane spaced very close to the split plane can provide high interplane capacitance and therefore serve as a low-impedance path for the return current. Practical economic concerns often require engineers to employ nonideal approaches and still deliver a working system. An isolated split plane requires more careful trace layout to absolutely minimize the number of signals that cross the break. When signals must cross a break on a layer adjacent to the split plane, the return path discontinuity can be minimized by placing bypass capacitors across the break in close proximity to the traces. The mechanism at work here is the same where a signal changes layers through a via. If an expliic \A—ow88H through the thin drawbridge. A fully isolated island requires power and ground to be supplied through passive filters or some other connection.

Remember that a key goal in designing for signal integrity is to minimize loop area and return path discontinuities so that less energy is radiated from a wire when it is driven, and less energy is picked up as noise when other wires in the system are being driven.

18.5 GROUNDING AND ELECTROMAGNETIC COMPATIBILITY

By now it should be clear that grounding is a critical aspect of system design. Grounding becomes more important as speeds increase, because more intense electromagnetic fields are present, and higher frequencies radiate more efficiently from smaller antennas. *Electromagnetic compatibility* (EMC) is the ability of a system to peacefully coexist with other systems so that it neither malfunctions because of excessive EMI susceptibility nor causes other systems to malfunction as a result of excessive electromagnetic field radiation. In most situations, EMC means being a good neighbor and complying with governmental regulations on how much electromagnetic energy an electronic system can radiate. The Federal Communications Commission governs such regulations in the United States. Most digital systems applications are not particularly sensitive to ambient electromagnetic energy. The chances are pretty low that a computer will malfunction during normal use because of excessive ambient fields. Of course, this does not hold true in some demanding applications such as aerospace and military electronics.

Our discussion is concerned with basic techniques for reducing a system's radiated electromagnetic emissions in the context of complying with government regulations. EMI reduction through minimizing loop area and removing return path discontinuities is a fundamental starting point for EMC. Reasonable steps should be taken up front to minimize the energy that a circuit board radiates. If a sloppy design radiates significant energy, it may be difficult or impossible to effectively contain these fields to the point of regulatory compliance.

Electromagnetic energy can escape from a circuit board by radiating into space or conducting onto a cable. Radiated emissions can be blocked by enclosing the circuit board in a grounded metal enclosure. This is why many computers and other electronic equipment have metal chassis, even though the metal may be hidden under a plastic frame or bezel. Most metal enclosures are not perfect closed surfaces, because slots and holes are necessary for cables, switches, airflow, and so on. Enclosures also must be assembled and are often opened for service, so there are numerous seams, hinges, and joints that connect one sheet of metal to another. All openings in the metal are potential leakage points for radiation, depending on their size. A hole forms a slot antenna whose efficiency is a function of its size and the wavelength, λ, of energy being radiated. When engineers construct antennas, $\lambda \div 4$ and $\lambda \div 2$ are typical dimensions that radiate most efficiently. Clearly, slots and holes whose largest dimensions approach $\lambda \div 4$ are undesirable. Limiting chassis openings to be substantially smaller than $\lambda \div 4$, perhaps $\lambda \div 20$, is necessary.

The most troublesome gaps that cause EMC problems are improperly grounded connector shells and poorly fitted seams between metal surfaces. Many connectors are available with metal bodies or metal shields around plastic bodies. If these metal surfaces are securely mounted to the metal chassis at multiple points such that openings are substantially smaller than $\lambda \div 4$, there should be no gap at the connector to allow excessive radiation. Seams between individual sheets of metal, either movable or fixed panels, must be designed to meet cleanly without the slightest buckling. Imperceptible buckling at seams opens gaps that radiate unwanted energy. A well designed sheet metal chassis should have all of its fixed panels adequately riveted or welded to ensure uniform contact across the seam. Movable panels almost always require additional assistance in the form of conductive gaskets and springs. A gasket or spring serves as a flexible conductor that closes any electrical gaps between two metal surfaces that move over time and that may expand and contract with temperature changes. Gasketing is directly akin to the rubber washer in a sink faucet—minute gaps must be closed to prevent leakage.

Plastic enclosures can also be shielded by applying conductive coatings, although this is usually more expensive than a sheet metal chassis, which is a reason why many products use metal rather than all-plastic enclosures. Many small electronic products can get away with less expensive, more attractive, and lighter uncoated plastic packaging, because their circuits do not radiate excessive energy. This may be because of their relatively slow signals, careful circuit design, or a combination of both.

Unwanted high-frequency noise that may be conducted onto exterior cables should be filtered between the active circuitry and the cable connector. Passive differential and common-mode filters are discussed earlier in this book. High-frequency data interfaces often have standard means of dealing with noise, including application-specific off-the-shelf transformers and common-mode chokes. Lower-frequency interfaces such as RS-232 may be effectively filtered with a second-order LC filter using either a choke or ferrite bead for the inductive element. If noise is still able to couple onto internal wiring harnesses that lead outside the enclosure, the weapon of last resort may be a ferrite core. Ferrite cores are available in clamshell types whereby the ferrite fits around a cable, and in ring forms whereby the cable is wrapped several turns around the core. The ferrite increases the inductance of the cable, which increases its attenuation of high frequencies. When you see a computer monitor cable or some other type of cable that has a noticeable round bulge near one end, a clamshell ferrite has been added, because the equipment was unable to pass emissions regulations without it.

Filters can also be employed to attenuate higher-order harmonics of digital signals as they are distributed on the circuit board to reduce the strength of ambient electromagnetic fields on the board and within the enclosure. Clock distribution can account for a substantial fraction of unwanted emissions, especially at higher-order harmonics that radiate through small metal gaps. One technique is to insert lowpass filters at clock buffer outputs to attenuate energy beyond the fifth harmonic. A square wave substantially retains its characteristics with only the first, third, and fifth harmonics present. Unfortunately, component variation, mainly in capacitors, across the individual filters on a low-skew clock tree can introduce unwanted skew at the loads. Instead of an LC or RC filter at the source, inserting just a ferrite bead may provide sufficient high-frequency attenuation to substantially quiet a system. If it is unclear whether such filtering is necessary, the design can include ferrite beads as an option. Ferrite bead PCB footprints can be placed at each output of a clock driver in very close proximity to any series termination resistors that might already be in the design. If the ferrites are not needed, they can be substituted with 0-Ω resistors. Introduction of an extra 0-Ω resistor very close to the clock driver should not cause problems in most systems. For truly conservative situations, these scenarios can be modeled ahead of time with field-solver software.

The method by which a system's many ground nodes are connected has a major impact on EMC in terms of noise radiating from cables leaving the chassis. Conceptually, there is a single ground node that all circuits use as their reference. This is easy to achieve at DC, because resistance is the dominant characteristic that causes voltage drops, and solid sheets of metal have very low sheet resistances. Additionally, there are no EMC problems at DC, because there is no AC signal to radiate. Inductance becomes the problem in maintaining equipotential across an entire system's ground structure at high frequencies. Small voltage differences appear across a circuit board's ground plane despite its low sheet inductance. These differences can cause EMC problems despite having little to no effect on signal integrity. The ideal situation is to ground everything to the same point to achieve an equipotential ground node, but finite physical dimensions make this impossible.

Any opportunity for a cable to have a high-frequency potential difference with respect to the chassis is an opportunity for unwanted electromagnetic radiation. The basic idea in many systems is to take advantage of a chassis' sheet metal surfaces as a clean ground reference because of low inductance and negligible current circulation. If a circuit board is grounded to one face of the chassis, and all cables are grounded to that same face, the ground potentials in that region will be nearly equal, with less opportunity for radiated emissions.

A complete discussion of chassis grounding techniques for EMC design is beyond the scope of this presentation. If you anticipate having to pass governmental electromagnetic emissions requirements, further reading is recommended. Electronic products are tested and certified for regulatory compliance at licensed test ranges where it is also common to find EMC consultants to advise you on solutions to emissions problems. Like most design tasks, it is better to seek help before building a product than to wait until a problem arises, at which point it is usually more expensive and time consuming to resolve.

18.6 ELECTROSTATIC DISCHARGE

Electrostatic discharge (ESD) is another phenomenon related to EMC and grounding. Static electric discharges are common occurrences and have been experienced by everyone. An insulated object accumulates a static electric charge and holds this charge until it comes into close proximity with a conductor. The human body can easily accumulate a 15,000-V charge while walking on carpet. If a person with a 15-kV charge comes into close proximity with a conductor at a substantially different potential (e.g., Earth ground), the charge may be able to arc across the air gap and discharge into that conductor. Higher potentials can jump across greater distances between the charged body and nearby conductors. The problems with ESD are twofold. First, ESD can disrupt a circuit's normal operation by inducing noise that causes errors in digital signals. Second, ESD can permanently damage components if the event is strong enough and the circuit is not protected. CMOS logic is particularly sensitive to ESD because of a FET's high gate impedance and the possibility of punching through the thin gate dielectric if a high potential is introduced.

When an ESD event occurs, it can couple onto a system's internal wires by inductive or capacitive means. A discharge is a brief, high-frequency, high-amplitude event with current peaking on the order of 10 A at 300 MHz. When ESD occurs, a very strong magnetic field is generated by the fast current spike. This field can be picked up by wires some distance away, and the coupling characteristics are governed by the same EMI concepts discussed earlier. Larger loops and thicker dielectrics make a more efficient antenna for ESD. A discharge to a chassis' metal panel not only establishes a strong magnetic field, it also creates a capacitor wherein the panel accepts the high-frequency signal and then may capacitively couple this energy to nodes within the enclosure. ESD occurs so rapidly that normal ground wires have too much inductance to drain the charge before it can do damage. A typical chassis is grounded to Earth through the AC power cord. This connection prevents gradual charge accumulation to dangerous potentials, but it cannot be expected to drain ESD before a circuit is disrupted.

This page intentionally left blank.

CHAPTER 19
Designing for Success

A host of details allow the major components of a system to function properly. Miscellaneous topics are often left out of many engineering discussions, because it is assumed that they will be covered elsewhere. This chapter attempts to gather into one place some of the remaining practical issues that make the difference between a smooth development process and one that is punctuated by a series of obstacles that waste time and detract from the operation of unique design elements that represent a system's true value.

Acquiring the necessary components and fabricating circuit boards is a mandatory step between design and testing. It is important to select technologies that are appropriate for both the application and your own resources. Practical considerations such as business relationships and support costs may constrain the choices of components and materials at your disposal. In extreme cases, it may not be possible to realize certain design goals with limited resources. In other situations, alternative

19.1 PRACTICAL TECHNOLOGIES

The combined semiconductor, electronics, and packaging industries develop many exciting and advanced technologies each year. An engineer may be tempted to use the latest and greatest components and assembly techniques on a new project, but careful consideration should go into making such decisions. Relevant constraints for any engineering organization are materials availability and cost, ease of manufacture, development resources, and general risk assessment. These constraints differ among organizations. A large company with extensive experience and support resources has a different view of the world from that of a one-man design shop building microcontrollers. This does not mean that a small organization cannot successfully utilize new technology. It does mean that all organizations must evaluate the practicality of various technologies using constraints that are appropriate for their size and resources.

Materials availability is often a problem, even for large companies, when dealing with cutting-edge technologies or newly introduced products. Cutting-edge technology is, by definition, one that pushes the limits of what is achievable at any given time. Pushing the limits in any discipline generally carries with it the understanding that problems may arise in the early stages of product release. New technologies may also carry higher initial costs while volumes and manufacturing yields are still low. Part of engineering is balancing the risks and benefits of new technologies. When you move into uncharted waters, an occasional setback is almost inevitable. Therefore, the new technology that one may read about in the trade press or see advertised in company literature is not necessarily ready for immediate use.

Aside from the general risk of new products, the economic strength that you represent as a customer has a significant impact on your ability to gain access to these products. If you are a semiconductor company that has just developed a new chip, and you have the staff to support only three initial customers, would you want three large customers or three small customers? Developing relationships with manufacturers and their representatives can assist you in determining when a new technology is practical to use and when it should be allowed to mature further. This applies equally to more mature products. Even components that have been shipping for some time may be subject to availability problems. The term *allocation* is well known to component buyers. In a tight market, vendors will preallocate their manufacturing capacity across a set of key customers to preserve successful business relationships. Even when a product is mature and being manufactured in high volume, a small customer may be unable to purchase it, because it is "on allocation." Allocation problems affect large companies as well in times of increased demand. The semiconductor industry tends to be quite cyclical, moving through phases of supply shortages and softness in demand.

Evaluating the risk of availability is an important step in the component selection process. More mature components are generally easier to obtain. The exceptions to this rule are ICs that have short production lives, such as some microprocessors and memory chips—especially DRAM. The microprocessor and memory markets are highly competitive, and products are sometimes phased out after just a few years. DRAM products are notorious for supply and obsolescence problems after their volumes peak within the first few years of introduction. There are certain bread-and-butter microprocessor and memory ICs that are supported for longer terms. These tend to be products for embedded markets in which semiconductor process technology changes at a slower pace than in the mainstream computer market.

Newer products are often available only through authorized distribution companies. Many mature products can be purchased from catalog distributors. Catalog distributors include Digikey, Jameco Electronics, JDR Microdevices, and Mouser Electronics. Larger engineering organizations with dedicated purchasing staff often prefer to deal with authorized distributors because of more direct access to manufacturers. A small organization may be able to satisfy all or most of its procurement needs with catalog distributors if mature technologies are acceptable.

ble so that logic mistakes do not require replacing a potentially expensive IC. Smaller, more mature PLDs may still require dedicated programming hardware, which may be reason enough to avoid them if possible in favor of a small CPLD. The price difference between a PLD and a small CPLD is now slim to none.

For reasons previously discussed, signal integrity software packages may be necessary when designing high-speed digital circuits. These tools can be quite expensive, but the consequences of not using them can be even more costly in terms of wasted materials and time if a circuit malfunctions because of signal integrity problems. Before embarking on an ambitious high-speed design, make sure that signal integrity issues are either well understood or that the resources are available for proper analysis before fabricating a prototype. Certainly, not all designs require extensive signal integrity analysis. If it is known that the signal speeds and wire lengths are such that transmission line effects, crosstalk, and EMI can be addressed through conservative design practices, minimal analysis may be required. This determination generally requires someone with prior experience to review a design and make predictions based on previous work.

Risk assessment in choosing which components and technologies to employ is an important part of systems design. An otherwise elegant architecture can fall on its face if a key component or necessary development tools are unavailable. Therefore, be sure to make choices that are practical for both the application and the resources at your disposal.

19.2 PRINTED CIRCUIT BOARDS

The selection of appropriate technologies is a convenient segue into circuit construction, because the manner in which a circuit is assembled can have a great impact on the viability of the resulting prototype or product. Higher-speed circuits are more sensitive to construction techniques because of grounding and inductance issues. Most high-speed circuits can be fabricated only with multilayer PCBs, but more options are available for slower systems, especially in the prototyping phase of a project.

Circuit boards can be of either the printed circuit or manual point-to-point wiring variety. As already discussed, PCBs consist of stacked layers of copper foil that have been uniquely etched to connect arbitrary points in the circuit. The term “printed” refers to the standard technique of using photolithography to expose a chemically treated copper foil with a negative image of the desired etching. Similar to creating a photograph, the exposure process alters the photoresistive chemicals that have been applied to the foil so that the exposed or nonexposed areas are etched away when the foil is placed into a chemical acid bath. PCBs are an ideal technology, because they can be mass produced with fine control over the accuracy of each unit. Simple single- and double-sided PCBs can be manually fabricated using a variety of techniques, and the cost of having such boards professionally manufactured is low. Multilayer PCBs must be fabricated professionally because of the complexity of creating plated vias and accurately aligning multiple layers that are etched separately and then glued together. The major cost involved in designing a small PCB is often the specialized computer aided design (CAD) software necessary to create the many features that a PCB implements, including accurate traces, pads, and IC footprints. Low-end PCB CAD packages are available for several hundred dollars. High-end tools run into the tens of thousands of dollars.

Once a PCB is fabricated, it is assembled along with the various components to which it is designed to connect. Assembly may be performed manually or at a specialized assembly firm, almost all of which use automated assembly equipment. It is difficult to manually assemble all but relatively simple boards because of the fine-pitch components and the element of human error. Automated assembly equipment substantially increases reliability and improves assembly time for multiple boards, but at the expense of increase setup time to program the machines for a specific design.

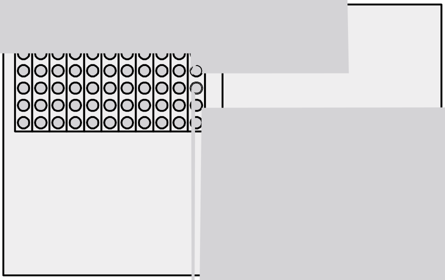
all components. These keep-outs make the PCB easier to handle and allow it to ride on rails through the pick-and-place, wave, and reflow machines. A typical component-to-edge keep-out is 0.2 in (5 mm). Very dense PCBs that cannot tolerate such keep-outs may require snap-off rails fabricated as part of the PCB. Such rails are not uncommon and are almost free, because they are formed by routing slots at the edges of a PCB as shown in Fig. 19.1. A related assembly rule is the inclusion of tooling holes at several locations around the PCB perimeter. These holes provide alignment and attachment points for the assembly machines.

Pick-and-place machines generally require assistance in perfectly aligning a high-pin-count SMT package to its designated location on the PCB. Most passive components and small multilead SMT

quency bulk electrolytic capacitors are arranged throughout the tains 7400-type logic ICs in 14-, 16-, and 20-pin DIPs as well as Cs in 28- and 40-pin DIPs.

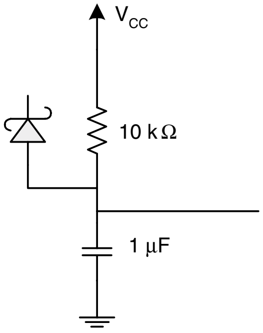
eadboards for temporary prototypes of small circuits is the *sold-* eadboard isn't a fiberglass board, but a plastic frame in which ed. Holes are on 0.1-in centers, and the spring clips are typically eparated by a gap, or channel, across which a standard 0.3-in gure 19.5 shows a small solderless breadboard. Power distribu- continuous spring-clip bus running across the top and bottom of de by inserting solid wires between various spring clips. Since maximum of five connections can be made to a single node. If spring clip nearby must be used for the excess connections and via a wire. Solderless breadboards are perfect for small experi- c lab settings, because solder irons and other assembly tools are eadboards are not for every circuit. Aside from electrical integ- non PLCC packages without a special breakout product that es- Nevertheless, substantial digital circuits can be prototyped on a -speed microprocessors with memory and basic peripherals.

prototype a digital system with permanent connections. *Wire-* n around for decades and was actually used for production as- mainframes during the 1960s and 1970s. The wire-wrapping ions by tightly wrapping small wire, typically 30 gauge, around made as shown in Fig. 19.6, resulting in a surprisingly durable without the benefit of insulation that is often stripped off. Wire- C sockets with long, square posts that protrude through the bot-



tom side of a breadboard. Wire-wrap sockets are still available for DIPs in standard sizes for two or three wraps per post. The actual wrapping is accomplished with a special tool—either manual or automatic.

A benefit of wire-wrapping is that dense wiring can be achieved without the risk of melting through insulation with a hot soldering iron. Changes in connectivity are made by unwrapping a wire. However, if the wire to be unwrapped is at the bottom of a stack of other wires, those others may need to be removed as well. A fully manual wire-wrap process requires that a wire be cut to length and stripped at each end to expose approximately 1 in of bare wire, and the wrapping tool is



time constant, as is required by some microprocessors. Better results are obtained using a Schottky diode, because its lower forward voltage discharges the capacitor to a lower voltage.

The second circuit is more robust, because it uses a Schmitt trigger to drive the microprocessor's input, guaranteeing a clean digital transition despite variations in the slope of the RC voltage curve. This is especially helpful when long RC time constants are required to generate long reset pulses as dictated by a microprocessor. A 74LS14 or similar Schmitt-trigger logic gate may be convenient to design into a system, and it can be used in places other than the power-on-reset circuit. Alternatively, a smaller voltage comparator can be used to implement the same function by designing in hysteresis. Before power-on, the inverter input node is at 0 V. At power-on, the voltage step of the power supply passes through the capacitor, because the voltage across the capacitor is initially 0 V and brings the input node to a logic-1 voltage, which in turn causes RESET* to be driven to logic 0. The resistor immediately begins pulling the voltage toward ground and eventually causes RESET* to be deasserted. The diode is present to clamp the inverter input node to ground during power-down. Clamping is desirable, because the resistor has already pulled the input node to ground and, without the diode, a negative V_{CC} step would force the input node to a negative voltage. When the diode is present, the input node remains near 0 V and is able to serve its intended purpose during an immediate power-on. The diode also prevents a large negative voltage from potentially damaging the inverter.

Power-on is not the only condition in which a microprocessor reset may be desired. Especially during the debugging process, it can be very useful to have a reset button that can quickly restart the system from a known initial state when software under development encounters fatal bugs. Many of the aforementioned power-on-reset ICs contain circuitry to *debounce* an external pushbutton. When a button is pushed, it may appear that a clean electrical connection is made and then broken when the button is released. In reality, the contact and release events of a button are noisy for brief periods of time as the internal metal contacts come into contact with each other. This noise or bounce may last only a few milliseconds, but it can cause a microprocessor to improperly exit its reset state. Debouncing is the process of converting the noisy edges of a pushbutton into a clean pulse. Filtering is a general solution for debouncing a noisy event and can be performed in an analog fashion or digitally by taking multiple samples of the event and forcing the bouncing samples to one state or the other.

19.5 DESIGN FOR DEBUG

"To err is human" is a truism that directly applies to engineering. The engineering process is a combination of design and debugging in which inevitable problems in the original implementation are detected and fixed. Hardware problems include logic mistakes, incorrect circuit board connections, and improperly assembled systems. Regardless of its nature, a problem must be isolated before it can be corrected. Bugs can be hard to find in a digital system, because time is measured in nanoseconds, and logic 1s and 0s are not visually distinct as they flow across wires. The debugging process is made easier by employing specific design elements and methodologies that improve visibility into a system's inner workings.

A basic debugging aid is the ability to probe a digital circuit with an oscilloscope or logic analyzer so that the state of individual wires can be observed. (A logic analyzer is a tool that rapidly captures a set of digital signals and then displays them for inspection. Test equipment is discussed in more detail later on.) Some circuits require little or no modification to gain this visibility, depending on their density, packaging technology, and operating speed. A circuit that uses DIPs or PLCCs exclusively can be directly probed with an oscilloscope probe, and clip-on logic analyzer adapters may be used to capture many digital signals simultaneously. As circuits get denser and use fine-pitch surface mount ICs, it becomes necessary to use connectors specifically designed for logic analyzer attachment. The correct type of connector should be verified with your logic analyzer manufacturer. Using logic analyzer connectors provides access to a set of predetermined signals fairly rapidly, because a whole connector can be inserted or removed at one time rather than having to use individual clips for each signal.

Logic analyzer probing becomes more of a challenge at high frequencies because of transmission line effects. Top-of-the-line logic analyzers are designed to minimally load a bus, and they include controlled impedance connectors to reduce unwanted side effects of probing. Depending on the trace lengths involved and the specific ICs, minimal impact is possible with speeds around 100 MHz. Careful PCB layout is essential for these situations, and it is desirable to minimize stubs created by routing signals to the connectors. At frequencies above 100 MHz, transmission line effects can rapidly cause problems, and special impedance matching and terminating circuitry may be necessary to achieve nonintrusive logic analyzer probing. Logic analyzer manufacturers have circuit recommendations that are specifically customized to their products.

The ideal scenario is to have logic analyzer visibility for every signal in a system. In reality, 100 percent visibility is not practical. More complex buses are more difficult to debug and, consequently, stand to benefit more from logic analyzer connectors. A simple asynchronous microprocessor bus, on the other hand, can be debugged with an inexpensive analog oscilloscope if a logic analyzer is not available. Whereas logic analyzers stop time and display a timing diagram of the selected signals over a short span of time, analog oscilloscopes usually do not have this persistence. The persistence problem can be addressed with a technique known as a *scope loop*. A scope loop is usually implemented in software but can be done in hardware as well, and it performs the same simple operation continuously so that an oscilloscope can be used to observe what has become a periodic event.

Debugging a basic microprocessor bus problem with a scope loop can be explained with a brief example. Figure 19.8 shows a portion of a digital system in which a RAM is connected to a microprocessor and enabled with an address decoder. In normal operation, the microprocessor asserts an address that is recognized by the decoder, and the RAM is enabled. One possible bug is an incorrectly wired address decoder. If the microprocessor is unable to access the RAM, the first thing to check is whether the RAM's chip select is being asserted. A single RAM access cannot be effectively observed on a normal analog oscilloscope, because the event may last well under a microsecond. Instead, a scope loop can be created by programming the microprocessor to continually perform RAM reads. The chip select can now be observed on an oscilloscope, because it is a periodic event. If the chip select is not present, the address decoder's inputs and output can be tested one by one to ensure proper connectivity. If the chip select is being asserted, other signals such as output enable or individual address bits can be probed. Oscilloscopes have at least two channels, and the second channel can be used to probe one other signal in conjunction with chip select to observe rela-

matching and low-skew buffers can be largely ignored. Of course, signal integrity considerations still apply so that TCK is delivered without excessive distortion to each IC. TRST* is active low and allows supporting devices to have their JTAG logic restarted. TMS is active high and places an IC into test mode by activating its JTAG controller.

JTAG scan chains are normally operated with special software designed to apply and check user-defined test patterns, or vectors. IC manufacturers who support JTAG provide boundary scan description language (BSDL) files that tell JTAG software how to interact with an IC. A standard procedure is to combine a PCB netlist (a file that lists each connection on the board in detail) with multiple BSDL files in a JTAG software package to come up with a set of test vectors. These vectors are then run through each board after it is assembled to verify connectivity between JTAG-equipped devices. Vendors of JTAG testing packages include Asset InterTech, Corelis, and JTAG Technologies. Verifying an assembled PCB with JTAG can save days of manual debugging when there may be problems in the assembly of BGA and fine-pitch components.

19.7 DIAGNOSTIC SOFTWARE

Software can help the debugging process by implementing scope loops, but the potential exists for much higher-level assistance. Special-purpose diagnostic programs are extremely helpful in detecting problems that could otherwise take much more time to isolate. The basic idea behind a diagnostic is for the software to thoroughly test elements of the hardware one at a time. A complete memory diagnostic, for example, would test every bit in every memory location. If the test fails, the nature of the failure provides valuable clues as to what is wrong. If there is a pattern to the failure, an improperly connected address or data bit may be the culprit. If the pattern is random, there may be a timing problem that causes marginal behavior over time or the device may be bad. Diagnostics are useful in several phases of development, including initial debug, extended reliability testing, troubleshooting damaged systems, and screening newly fabricated systems in either laboratory or manufacturing environments.

Address bus wiring problems result in aliasing or an unexpected displacement within the memory range. In a 16-bit addressing range, a disconnected A[15] would float to a default logic level and result in the lower and upper 32,768 locations overlapping. If A[15] and A[14] were shorted together, the two bits would have only two logic states instead of four, causing the middle 32k of the address space to overlap with the upper and lower 16k regions.

It may be useful to have multiple memory diagnostics to help with different phases of debugging. Some engineers like basic walking-ones and walking-zeroes patterns to quickly determine if any data bits are stuck. As their names imply, walking-ones and walking-zeroes tests set all bits in a word to one state and then walk the opposite state across each bit position. An eight-bit walking ones test could look like this: 00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000. These tests verify that each bit position can be independently set to a 0 or 1 without interference from neighboring bits.

Testing an entire memory after simple data bus wiring problems are resolved can be done efficiently by selecting a set of appropriate data patterns. Pattern selection depends on the diagnostic goals. Is it sufficient to verify only stuck address bits? Both stuck address and data bits? Or is it necessary to verify every unique bit in a memory array? Stuck data bits can be uncovered with a quick walking-ones and walking-zeroes test targeted to just a few memory locations. It is important to separate the write and read phases of a short diagnostic, and therefore use multiple memory locations, to guarantee that the microprocessor is not merely reading back bus capacitance. Isolating and verifying every bit in a memory array can be a more complex operation than first imagined, because specific memory design aspects at both the board and chip levels may require application specific patterns to achieve 100 percent coverage. A common memory diagnostic approach is to first test for stuck data bits in a quick test and then verify the address bus with a longer test and, in the process, achieve good, but incomplete, bit coverage. This level of coverage in concert with the testing that is performed by semiconductor manufacturers provides a high degree of confidence in the memory system's integrity.

Verifying address bus and decode logic integrity can be done with a set of ramp patterns. Each memory location is ideally written with a unique value, but this is not possible in systems where the data bus width is less than the address bus width. A microprocessor with 32-bit wide memory bus and 2^{20} locations can write a unique incrementing address into each word during a write phase and the verify on a subsequent read phase. A system with an 8-bit memory and 65,536 locations cannot write unique values in a single pass. An initial approach to verifying a 64-kB memory array is to write a repeating ramp pattern from 0x00 to 0xFF throughout memory and then read it back. This tests only the lower half of the address bus, because there are only 256 unique values written. Aliasing is a potential problem, because a 256-byte memory would appear identical to a 64-kB memory without further effort. The diagnostic requires a second pass to test the upper half of the address bus by writing the MSB of the address to each byte. Each pass is incomplete on its own, but together they are an effective diagnostic.

There are many memory diagnostic techniques, and engineers have their own favorites. The repeating, alternating pattern of 0x55 and 0xAA is popular, because it provides some verification that each bit position can hold a 1 or 0 independent of its nearest neighbors. This is in contrast to the pattern of 0x00 and 0xFF that could pass even if every data bit were shorted to every other data bit. However, 0x55 and 0xAA cannot isolate all data bus problems, because there is no restriction that shorts must occur between directly adjacent bits. Traces on a PCB are routed in many arbitrary patterns and shorts between nonadjacent bits are possible.

Each system requires a customized diagnostics suite, because each system has different memory and I/O resources that need to be tested. Serial communications diagnostics are common, because most systems have serial ports. A loop-back cable is connected to the serial port so that all data transmitted is received at the same time. The loop-back forms a complete data path that can be auto-

are effective. The degree to which a Spice simulation matches reality depends on how closely the real conditions are modeled. Performing highly accurate simulations is a skill that requires a thorough understanding of circuit theory. However, useful first-order approximations of analog behavior can be readily achieved. A key source of divergence between simulation and reality in a digital design is the parasitic properties of wires and components that become significant at high frequencies. An idealized resistor or wire might require the explicit addition of parasitic inductance and capacitance to get a more accurate simulation.

A basic example of Spice simulation can be shown using the first-order RC filter in Fig. 19.11. This lowpass filter uses idealized components and has $f_C \approx 10$ MHz with a steady attenuation slope of 20 dB per decade.

Circuits are presented to Spice by uniquely naming or numbering each node and then instantiating circuit elements that reference those node names. Figure 19.12 shows the Spice circuit description for the idealized RC filter. Ground is represented as 0. Resistors and capacitors are designated with identifiers beginning with R and C, respectively. V denotes a voltage source, and this voltage source is specified with a 0-V DC component and a 1-V AC component. The `.AC`

tion. As expected, there is greater distortion, because the transmission line is not as well terminated. Approximately 0.5 V of overshoot and undershoot are observed.

The preceding examples provide a brief glimpse of what is possible with Spice. In addition to passive circuits, semiconductors and active components can be modeled, enabling detailed simulations of analog amplifiers and digital circuits. Spice enables simulation to the desired level of resolution without forcing undue complexity. Small, quick simulations suffice for circuits with significant margins. More complex and detailed simulations that use highly accurate models are called for when operating near the limits of technology where margins for errors are very small.

19.9 TEST EQUIPMENT

Test equipment needs vary, depending on the complexity of a project. There are general types of test equipment used to debug and measure wide varieties of circuits, and there are very specialized tools designed for niche applications. This section provides a brief introduction to the more common pieces of test equipment found in typical engineering laboratories. As with any equipment, costs and capabilities vary widely. A 25-MHz analog oscilloscope may cost several hundred dollars, whereas a

passes. These DSOs take relatively few samples on each pass, which is why a signal must be repetitive so that multiple passes are sampling essentially the same waveform. Sampling resolution is a closely related characteristic. It is common to find DSOs with eight-bit resolution, which is adequate for many digital probing needs because of these signals' binary nature. Certain analog applications may require finer resolution to take proper measurements, and such improvements come at a cost.

A relative of the oscilloscope is the logic analyzer, a device that is intended for purely digital test applications. Like a DSO, a logic analyzer captures signals at high sampling rates as they occur and then freezes them for human analysis for an arbitrarily long time. The principal differences are that a logic analyzer captures single-bit samples and, consequently, is able to work with dozens or hundreds of channels at the same time. High channel count enables a logic analyzer to capture complex buses in their entirety so that a complete picture of a bus's digital state can be displayed. An oscilloscope can show that a write-enable is coinciding with a chip-select. A logic analyzer can also show the $8—A-e8fan°C*S8—An8—AHongurG x—HftOm x9O8tG }, B ofochipO8fB x9——fof88AcurOHe-uwx

APPENDIX A
Further Education

One of the exciting aspects of electrical engineering is that the state of the art changes quickly. Consequently, there is always the need to learn about new technologies, methods, and components. The modern engineer is fortunate to have a multitude of educational resources from which to draw. The Internet has made technical information more accessible than ever to anyone with a modem. Educational resources for engineers include

- Trade publications and subscription periodicals
- Technical books
- Manufacturers' web sites and publications
- Third-party web sites
- Colleagues and conferences

Trade publications and other periodicals are a good way to keep aware of current trends in the industry, because articles are often written about new and interesting technologies. Even advertisements provide an education, because manufacturers' claims can be compared against each other and against information acquired elsewhere. Many trade publications are funded by advertisements and can therefore offer free subscriptions to qualified subscribers with relevant professional responsibilities and technical needs. The following periodicals are recommended by the author:

- *Circuit Cellar* (paid subscription). The focus is on embedded systems with articles describing real implementations and discussions of how to make practical systems work.
- *EDN* and *Electronics Design* (free subscriptions). Articles are written by staff editors and professional engineers on topics from current trends to specific design implementation techniques and technical advice.
- *EE Times* (free subscription). This is a general electronics industry weekly news with articles on current trends and new technologies.

Technical books are available on practically every aspect of engineering. Chances are that you are already aware of this option, because you are reading this book right now! The following books are recommended as references for various topics that arise in digital electrical engineering:

- Computer architecture—*Computer Organization and Design: The Hardware/Software Interface,* David A. Patterson and John L. Hennessy, Morgan Kauffman.
- Signal integrity and PCB design—*High-Speed Digital Design: A Handbook of Black-Magic,* Howard W. Johnson and Martin Graham, Prentice Hall.

INDEX