

THE **RT** OF ELECTRONICS

**PAUL HOROWITZ
WINFIELD HILL**

**THIRD
EDITION**

The Art of Electronics

Third Edition

At long last, here is the thoroughly revised and updated, and long-anticipated, third edition of the hugely successful *The Art of Electronics*. Widely accepted as the best single authoritative text and reference on electronic circuit design, both analog and digital, the first two editions were translated into eight languages, and sold more than a million copies worldwide. The art of electronics is explained by stressing the methods actually used by circuit designers – a combination of some basic laws, rules of thumb, and a nonmathematical treatment that encourages understanding why and how a circuit works.

Paul Horowitz is a Research Professor of Physics and of Electrical Engineering at Harvard University, where in 1974 he originated the Laboratory Electronics course from which emerged *The Art of Electronics*. In addition to his work in circuit design and electronic instrumentation, his research interests have included observational astrophysics, x-ray and particle microscopy, and optical interferometry. He is one of the pioneers of the search for intelligent life beyond Earth (SETI). He is the author of some 200 scientific articles and reports, has consulted widely for industry and government, and is the designer of numerous scientific and photographic instruments.

Winfield Hill is by inclination an electronics circuit-design guru. After dropping out of the Chemical Physics graduate program at Harvard University, and obtaining an E.E. degree, he began his engineering career at Harvard's Electronics Design Center. After 7 years of learning electronics at Harvard he founded Sea Data Corporation, where he spent 16 years designing instruments for Physical Oceanography. In 1988 he was recruited by Edwin Land to join the Rowland Institute for Science. The institute subsequently merged with Harvard University in 2003. As director of the institute's Electronics Engineering Lab he has designed some 500 scientific instruments. Recent interests include high-voltage RF (to 15 kV), high-current pulsed electronics (to 1200 A), low-noise amplifiers (to sub-nV and pA), and MOSFET pulse generators.

This page intentionally left blank

THE ART OF ELECTRONICS

Third Edition

Paul Horowitz HARVARD UNIVERSITY

Winfield Hill ROWLAND INSTITUTE AT HARVARD

CAMBRIDGE
UNIVERSITY PRESS

32 Avenue of the Americas, New York, NY 10013-2473, USA

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/9780521809269

© Cambridge University Press, 1980, 1989, 2015

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 1980

Second edition 1989

Third edition 2015

7th printing 2016 with corrections

Printed in the United States of America

A catalog record for this publication is available from the British Library.

ISBN 978-0-521-80926-9 Hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet websites referred to in this publication and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

10.1 Basic logic concepts

10.1.1 Digital versus analog

Thus far we have been dealing mainly with circuits in which the input and output voltages have varied over a continuous range of values: *RC* circuits, amplifiers, integrators, rectifiers, op-amps, etc. This is natural when dealing with signals that are continuous (e.g., audio signals) or continuously varying voltages from measuring instruments (e.g., temperature-reading or light-detecting devices, or biological or chemical probes).

However, there are instances in which the input signal is naturally discrete in form, e.g., pulses from a particle detector, or “bits” of data from a switch, keyboard, or computer. In such cases the use of digital electronics (circuits that deal with data made of 1s and 0s) is natural and convenient. Furthermore, it is often desirable to convert continuous (analog) data to digital form, and vice versa, using analog-to-digital converters (ADCs) and digital-to-analog converters (DACs), in order to perform computations on the data (with a computer or signal processor) or to store large quantities of data as numbers. In a typical situation a microprocessor or computer might monitor signals from an experiment or industrial process, control the experimental parameters on the basis of the data obtained, and store for future use the results collected or computed while the experiment is running.

Another interesting example of the power of digital techniques is the transmission of analog signals without degradation by noise: an analog audio or video signal, for instance, picks up “noise” while being transmitted by cable or wireless that cannot be removed. If, instead, the signal is converted to a series of numbers representing its amplitude at successive instants of time, and these numbers are transmitted as digital signals, the analog signal reconstruction at the receiving end (done with DACs) will be without error, providing the noise level on the transmission channel isn’t great enough to prevent accurate recognition of 1s and 0s. This technique, known as PCM (pulse-code modulation), is particularly attractive where a signal must pass through a series of “repeaters,” since digital regeneration at each

stage guarantees noiseless transmission. The information and stunning pictures sent back by planetary deep space probes, for example the Pioneer 10 mission to Jupiter in 1973, were stored and transmitted with PCM. Digital audio and video are now commonplace in the home, for example in the form of the humble 12 cm optical CD,¹ in which music is stored in the form of a stereo pair of 16-bit numbers every 23 microseconds, (1.4 megabits/sec), 6 billion bits (gigabits, Gb) of information in all. And by contemporary standards this is a storage medium of low speed and capacity: the corresponding figures for DVDs and Blu-rays are 10 and 48 megabits/sec (maximum), with a total storage of 38 and 200 gigabits per layer.

In fact, digital hardware has become so powerful that tasks that would have seemed well suited to analog techniques are usually better solved with digital methods. As an example, an analog temperature meter might incorporate a microprocessor and memory to improve accuracy by compensating for the instrument’s departure from perfect linearity; likewise for something as commonplace as a digital bathroom scale. Because of the wide availability of inexpensive (less than \$1) microcontrollers, such applications are ubiquitous. The average home is awash in devices with “embedded” processors: in every music player, television set, cellphone, dishwasher, washing machine and dryer, fax, copier, microwave oven, coffee maker, . . . The list is long, and sometimes surprising.² Rather than attempt to enumerate what can be done with digital electronics, let’s just start learning about it. Applications will emerge naturally as we go along.

¹ And their video counterparts: DVDs and Blu-ray (BD) discs.

² One might encapsulate the history of the “conquest of digital over analog” something like this: the 1970s – planetary photographs sent digitally from 800 million kilometers away, but domestic life still ruled by analog; the 1980s – digital audio (CDs) and the personal computer invade the home; the 1990s – digital video (DVDs), cellphones, and MP3s; the 2000s – HDTV, digital wireless connectivity, digital photography and videography, high-speed Internet, and *Google*; and the 2010s (OK, OK, we know the old saying “It’s difficult to predict . . . especially the future”) – universal digital dissemination and fusion of information and media.

10.1.2 Logic states

By “digital electronics” we mean circuits in which there are only two (usually) states possible at any point, e.g., a transistor that can either be in saturation or be nonconducting. We usually choose to talk about voltages rather than currents, calling a level HIGH or LOW. The two states can represent any of a variety of “bits” (binary digits) of information, such as the following:

- one bit of a number;
- whether a switch is opened or closed;
- whether a signal is present or absent;
- whether some analog level is above or below some preset limit;
- whether or not some event has happened yet;
- whether or not some action should be taken;
- and so on.

A. HIGH and LOW

The HIGH and LOW voltage states represent the TRUE and FALSE states of Boolean logic, in some predefined way. If at some point a HIGH voltage represents TRUE, that signal line is called “active HIGH” (or “HIGH true”) and vice versa. This can be confusing at first. Figure 10.1 shows an example. SWITCH CLOSED is true when the output is LOW; that’s an “active-LOW” (or “LOW-true”) signal, and you might label the lead as shown (a bar over a symbol means NOT; that line is HIGH when the switch is *not* closed). Just remember that the presence or absence of the negation bar over the label tells whether the wire is at a LOW or HIGH voltage state when the stated condition (SWITCH CLOSED) is true.³ At first the idea of active-LOW may seem, well, *backward*; why not just keep it simple and outlaw this upside-down logic? As we’ll see, though, there are good reasons to do things “backwards” sometimes. Be patient.

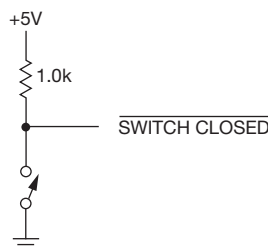


Figure 10.1. A LOW-true (“active-LOW”) logic level.

A digital circuit “knows” what a signal represents by where it comes from, just as an analog circuit might know what the output of some op-amp represents. However, added flexibility is possible in digital circuits; sometimes the same signal lines are used to carry different kinds of information, or even to send it in different directions, at different times. To do this “multiplexing,” additional information must also be sent (address bits, or status bits). You will see many examples of this very useful ability later. For now, imagine that any given circuit is wired up to perform a predetermined function and that it knows what that function is, where its inputs are coming from, and where the outputs are going.

To lend a bit of confusion to a basically simple situation, we introduce 1 and 0. These symbols are used in Boolean logic to mean TRUE and FALSE, respectively, and are sometimes used in electronics in exactly that way. Unfortunately, they are also used in another way, in which 1 = HIGH and 0 = LOW! In this book we try to avoid any ambiguity by using the word HIGH (or the symbol H) and the word LOW (or the symbol L) to represent logic states, a method that is in wide use in the electronics industry. We use 1 and 0 only in situations where there can be no ambiguity.

B. Voltage range of HIGH and LOW

In digital circuitry, the voltage levels corresponding to HIGH and LOW are allowed to fall in some range, according to the particular logic family.⁴ For example, with high-speed CMOS (“HC” family) logic running from a +5 V supply, *input* voltages within about 1.5 volts of ground are interpreted as LOW, whereas voltages within 1.5 volts of the +5 V supply are interpreted as HIGH.⁵ Those inputs are driven from the outputs of some other devices, for which typical LOW- and HIGH-state *output* voltages are usually within a tenth of a volt of 0 and +5 V, respectively (the output is a saturated transistor switch to one of the rails; see Figure 10.25). This allows for manufacturing spread,

⁴ A “family” is a particular hardware implementation of digital logic, characterized by operating voltage, logic voltage levels, and speed. For historical reasons, most logic families name their standard logic parts with a prefix of 74, followed by a few letters that name the family, and ending with some numbers that specify the logic function. The logical functions themselves are the same across families. For example, a 74LVCO8 is a 2-input AND gate (actually, four of them in one package) in the low-voltage CMOS (LVC) electrical family; the “08” designates a quad 2-input AND, and the “74” designates a logic chip for operation at standard temperatures.

⁵ See the box “Logic Levels” for additional examples.

³ You’ll sometimes see the terms “positive-true” and “negative-true” used for HIGH-true and LOW-true, respectively. Those terms are OK, but they can be confusing to the inexperienced, especially since no negative voltages are involved.

variations of the circuits with temperature, loading, supply voltage, etc., and the presence of “noise,” the miscellaneous garbage that gets added to the signal in its journey through the circuit (from capacitive or inductive coupling, external interference, etc.). The circuit receiving the signal decides if it is HIGH or LOW and acts accordingly.⁶ As long as noise does not change 1s to 0s, or vice versa, all is well, and any noise is eliminated at each stage, where “clean” 0s and 1s are regenerated. In that sense, digital electronics is noiseless and perfect.

The term *noise immunity* is used to describe the maximum noise level that can be added to logic levels (in the worst case) while still maintaining error-free operation. As an example, the formerly popular family of logic known as “TTL” (transistor–transistor logic) struggled with just this issue, because it has just 0.4 V of noise immunity: a TTL *input* is guaranteed to interpret anything less than +0.8 V as LOW and anything greater than +2.0 V as HIGH, whereas the worst-case *output* levels are +0.4 V and +2.4 V, respectively (see the accompanying box on logic levels). In practice, noise immunity will be better than the worst-case 0.4 V margin, with typical LOW and HIGH voltages of +0.2 V and +3.4 V and an input decision threshold near +1.3 V. But always remember that if you are doing good circuit design, you use worst-case values. It is worth keeping in mind that different logic families have different amounts of noise immunity. CMOS has greater voltage-noise immunity than TTL, whereas the speedy ECL (emitter-coupled logic) family has less. Of course, susceptibility to noise in a digital system depends also on the amplitude of noise that is present, which in turn depends on factors such as output-stage stiffness, inductance in the ground leads, the existence of long “bus” lines, and output slew rates during logic transitions (which produce transient currents, and therefore voltage spikes on the ground line, because of capacitive loading). We will worry about some of these problems in Chapter 12 (Logic Interfacing).

10.1.3 Number codes

Most of the conditions we listed earlier that can be represented by a digital level are self-explanatory. How a digital

⁶ Sometimes digital signals are sent as a differential voltage pair, rather than “single-ended.” This is particularly popular with longer high-speed signals, or signals that go off-board over some distance, for example fast serial buses such as USB, Firewire, and SATA; it is also commonly used to distribute high-frequency clock signals. A popular format is “LVDS” (low-voltage differential signaling), in which the differential signal amplitude is ~ 0.3 V, centered at +1.25 V.

level can represent part of a number is a more involved, and very interesting, question. Put another way, we’ve seen *bits as indicators*; now we will see *groups of bits as a number*.

A decimal (base-10) number is simply a string of integers that are understood to multiply successive powers of 10, the individual products then being added together. For instance,

$$137.06 = 1 \times 10^2 + 3 \times 10^1 + 7 \times 10^0 + 0 \times 10^{-1} + 6 \times 10^{-2}.$$

Ten symbols (0 through 9) are needed, and the power of 10 each multiplies is determined by its position relative to the decimal point. If we want to represent a number using two symbols only (0 and 1), we use the *binary*, or base-2, number system. Each 1 or 0 then multiplies a successive power of 2. For instance,

$$\begin{aligned} 1101_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 13_{10}. \end{aligned}$$

The individual 1s and 0s are called “bits” (binary digits). The subscript (always given in base 10) tells what number system we are using, and often it is essential in order to avoid confusion, since the symbols all look the same.

We convert a number from binary to decimal by the method just described. To convert the other way, we keep dividing the number by 2 and write down the remainders. To convert 13_{10} to binary, therefore:

$$\begin{array}{ll} 13/2 = 6 & \text{remainder } 1, \\ 6/2 = 3 & \text{remainder } 0, \\ 3/2 = 1 & \text{remainder } 1, \\ 1/2 = 0 & \text{remainder } 1, \end{array}$$

from which $13_{10} = 1101_2$. Note that the answer comes out in the order LSB (least significant bit) to MSB (most significant bit).

A. Hexadecimal (“hex”) representation

The binary-number representation is the natural choice for two-state systems (although it is not the only way; we’ll see some others soon). Because the numbers tend to get rather long, it is common to write them in hexadecimal (base-16) representation: each position represents successive powers of 16, with each hex symbol having a value from 0 to 15 (the symbols A–F are assigned to the values 10–15). To write a binary number in hexadecimal, just group it in 4-bit groups, beginning with the LSB, and write the hexadecimal equivalent of each group:

$$707_{10} = 1011000011_2 (= 10 \ 1100 \ 0011_2) = 2C3_{16}.$$

LOGIC LEVELS

The diagram in Figure 10.2 shows the ranges of voltages that correspond to the two logic states (HIGH and LOW) for popular families of digital logic. For each logic family it is necessary to specify legal values of both output and input voltages corresponding to the two states HIGH and LOW. The shaded areas above the line show the specified range of output voltages within which a logic LOW or HIGH is guaranteed to fall, with the pair of arrows indicating typical output values (LOW, HIGH) encountered in practice. The shaded areas below the line show the range of input voltages guaranteed to be interpreted as LOW or HIGH, with the arrow indicating the typical *logic threshold* voltage, i.e., the dividing line between LOW and HIGH. In all cases a logic HIGH is more positive than a logic LOW. Table 10.1 and Figure 10.26 provide additional information about these families, a subject we'll see in more detail in Chapter 12.

The meanings of “minimum,” “typical,” and “maximum,” in electronic specifications are worth a few words

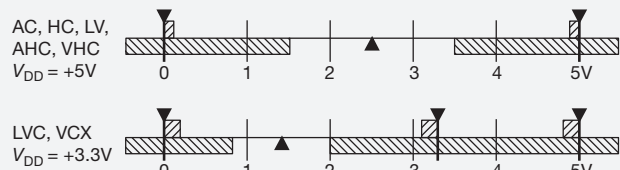


Figure 10.2. Logic levels of some popular logic families.

of explanation. Most simply, the manufacturer guarantees that the components will fall in the range minimum-to-maximum, with many close to “typical.” What this means is that typical specifications are an approximate guide to use when designing circuits; however, those circuits must work properly over the whole range of specifications from minimum to maximum (the extremes of manufacturing variability). In particular, a well-designed circuit must function under the worst possible combination of minimum and maximum values. This is known as *worst-case design*, and it is essential for any instrument produced from off-the-shelf (i.e., not specially selected) components.

Table 10.1 Selected Logic Families

Family	Manufacturers	Supply voltage (V _{CC})		V _{in} max	t _{pd} @ V _{CC} (ns, typ)		Available packages		
		min (V)	max (V)		(V)		DIP	SMT	1G, 2G
74HC00	9	2	6	V _{CC}	9	5	•	•	•
74AC00	5	3	6	V _{CC}	6	5	•	•	-
74AHC00	2	2	5.5	V _{CC}	3.7	5	•	•	•
74LV00	2	1.2 ^a	5.5	5.5	3.6	5	•	•	•
74LVC00	4	1.7	3.6	5.5	3.5	3.3 ^b	-	•	•
74ALVC00	3	1.7	3.6	3.6	2	3.3 ^c	-	•	-
74AUC00	1	0.8	2.7	3.6	0.9	1.8	-	•	•

Notes: (a) NXP only (TI specs to 2V only). (b) 6ns @ 1.8V. (c) 2.7ns @ 1.8V.

Hexadecimal representation⁷ is well suited to the popular “byte” (8-bit) organization of computers, which are most often organized as 16, 32, or 64-bit computer “words”; a word is then 2, 4, or 8 bytes. So in hexadecimal, each byte is 2 hex digits, a 16-bit word is 4 hex digits, etc. For example, the memory locations in a microcontroller with 65,536 (“64K”) bytes of memory can be identified by a 2-byte address, because $2^{16}=65,536$; the lowest address in hex is 0000h (the trailing “h” means hex), the highest address is FFFFh, the second half of memory begins at 8000h, and the fourth quarter of memory begins at C000h.

A byte sitting somewhere in computer memory can represent an integer number, or part of a number. But it can

also represent other things: for example, an alphanumeric character (letter, number, or symbol) is commonly represented as one byte. In the widely used ASCII representation (more in §14.7.8), lowercase “a” is represented as ASCII value 01100001 (61h), “b” is 62h, etc. Thus the word “nerd” could be stored in a pair of 16-bit words whose hex values are 6E65h and 7264h.

B. Binary-coded decimal

Another way to represent a number is to encode each decimal digit into binary. This is called BCD (binary-coded decimal), and it requires a 4-bit group for each digit. For instance,

$$137_{10} = 000100110111 \quad (\text{BCD}).$$

⁷ Alternative notations for a hexadecimal number (like 2C3₁₆) are 2C3H, 2C3h, 2c3h, and 0x2C3.

Note that BCD representation is *not* the same as binary representation, which in this case would be $137_{10} = 10001001_2$. You can think of the bit positions (starting from the right) as representing 1, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400, 800, etc. It is clear that BCD is wasteful of bits, because each 4-bit group *could* represent numbers 0 through 15, but BCD never represents numbers greater than 9. However, BCD is ideal if you want to display a number in decimal, since all you do is convert each BCD character to the appropriate decimal number and display it. For this reason, BCD is commonly used for input and output of numeric information. Unfortunately, the conversion between pure binary and BCD is complicated, because *each* decimal digit depends on the state of almost every binary bit, and vice versa. Nevertheless, binary arithmetic is so efficient that most computers convert all input data to binary, converting back only when data need to be output. Think how much effort and bother would have been saved if *Homo sapiens* had evolved with 8 (or 16) digits.⁸

Exercise 10.1. Convert to decimal: (a) 1110101.0110_2 , (b) $11.01010101\dots_2$, (c) $2A_H$. Convert to binary: (a) 1023_{10} , (b) 1023_H . Convert to hexadecimal: (a) 1023_{10} , (b) 101110101101_2 , (c) 61453_{10} .

Table 10.2 4-Bit Signed Integers in Three Systems of Representation

Integer	Sign-magnitude	Offset binary	2's comp
+7	0111	1111	0111
+6	0110	1110	0110
+5	0101	1101	0101
+4	0100	1100	0100
+3	0011	1011	0011
+2	0010	1010	0010
+1	0001	1001	0001
0	0000	1000	0000
-1	1001	0111	1111
-2	1010	0110	1110
-3	1011	0101	1101
-4	1100	0100	1100
-5	1101	0011	1011
-6	1110	0010	1010
-7	1111	0001	1001
-8	-	0000	1000
(-0)	1000	-	-

C. Signed numbers

Sign-magnitude representation

Sooner or later it becomes necessary to represent negative numbers in binary, particularly in devices where some computation is done. The simplest method is to devote one bit (the MSB, say) to the sign, with the remaining bits representing the magnitude of the number. This is called “sign-magnitude representation,” and it corresponds to the way signed numbers are ordinarily written (see Table 10.2). It is used when numbers are displayed, as well as in some ADC schemes. In general, it is not the best method for representing signed numbers (except for floating-point numbers), particularly where some computation is done, for several reasons: computation is awkward, and subtraction is different from addition (i.e., addition doesn’t “work” for signed numbers). Also, there can be two zeros (+0 and -0), so you have to be careful to use only one of them.

Offset-binary representation

A second method for representing signed numbers is “offset binary,” in which you subtract half the largest possible number to get the value represented (Table 10.2). This has the advantage that the number sequence from the most negative to the most positive is a simple binary progression, which makes it a natural for binary “counters.” The MSB still carries the sign information, and zero appears only once. Offset binary is popular in A/D conversions, but it is still awkward for computation.

2's complement representation

The method most widely used for integer computation is called “2’s complement.” In this system, positive numbers are represented as simple unsigned binary. The system is rigged up so that a negative number is then simply represented as the binary number you add to a positive number of the same magnitude to get zero. To form a negative number, first complement each of the bits of the positive number (i.e., write 1 for 0, and vice versa; this is called the “1’s complement”), then add 1 (that’s the “2’s complement”).⁹ As you can see from Table 10.2, 2’s complement numbers are related to offset binary numbers by having the MSB complemented. As with the other signed-number representations, the MSB carries the sign information. There’s only one zero, conveniently represented by all bits 0 (“clearing” a counter or register sets its value to zero). Because the 2’s complement system is natural for computation, (i.e., it allows computers to treat negative and

⁸ If the former, we’d be thumbless, thus spared the scourge of “texting.”

⁹ Alternatively, you can simply think of it as plain ol’ (unsigned) binary, but with the MSB representing the negative of its usual value.

positive integers the same way), it is universally used for integer arithmetic in computers.¹⁰

D. Arithmetic in 2’s complement

Arithmetic is simple in 2’s complement. To add two numbers, just add bitwise (with carry), like this:

5+(−2)=3:

0101	(+5)
+1110	(−2)
0011	(+3)

To subtract *B* from *A*, take the 2’s complement of *B* and add (i.e., add the negative):

2−5=−3:

0010	(+2)
+1011	(−5)
1101	(−3)

(Note in this last example that +5=0101, so its 1’s comp is 1010 and its 2’s comp is −5=1011.) Multiplication also “works right” in 2’s complement representation. Try the following exercises.

Exercise 10.2. Multiply +2 by −3 in 3-bit 2’s complement binary arithmetic. *Hint:* the answer is −6.

Exercise 10.3. Show that the 2’s complement of −5 is +5.

Because an *n*-bit integer can represent only 2^{*n*} numbers, you can get overflow or underflow when you add or subtract two numbers of fixed word size. To be precise, an *n*-bit unsigned integer can take on values from 0 to 2^{*n*−1}, and an *n*-bit 2’s complement signed integer can take on values from −2^{*n*−1} to +2^{*n*−1}−1. For 8-bit integers those ranges are 0 to 256, and −128 to +127. To determine if an unsigned add has overflowed (and is therefore incorrect), simply note if it generated a carry out of the MSB. For signed 2’s complement numbers the analogous criterion is slightly weird: if the sign bit (the MSB) is changed *by carries* (that is, if a carry in to the MSB is not balanced by a carry out, or vice versa), the result is incorrect.

Exercise 10.4. Check this unlikely criterion by doing a 2’s complement addition of each of the following signed-number pairs, assuming a 4-bit word size: 7 + (−6); 7 + 7; 7 + 4; −7 + (−8). Then repeat the same sums with a 5-bit word size, in which all the answers should fit.

E. Gray code

The following code is used for mechanical linear and shaft-angle encoders and for certain ADCs, among other things.

¹⁰ Note, however, that “floating-point” numbers are usually represented in a form of “sign magnitude,” namely, sign-exponent-mantissa.

It is called a Gray code,¹¹ and it has the property that only one bit changes in going from one state to the next. This prevents incorrect codes at the transitions, which otherwise produce errors because there is no way of guaranteeing that all bits will change simultaneously at the boundary between two encoded values. If straight binary were used, it would be possible to generate an output of 7 in going from 3 to 4, for example. Here is a simple rule for generating Gray-code states: begin with a state of all zeros. To get to the next state, always change the single least significant bit that brings you to a new state.

State	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Figure 10.3 shows how a Gray-coded angle encoder eliminates false codes at the transitions. Gray codes can be generated with any number of bits. They find use also in “parallel encoding” (also called *flash conversion*), a technique of high-speed A/D conversion that we will see later. We will talk about translation between Gray-code and binary-code representations in the next section (including a gate implementation in Figure 10.10).

10.1.4 Gates and truth tables

A. Combinational versus sequential logic

In digital electronics the name of the game is generating digital outputs from digital inputs. For instance, an *adder* might take two 16-bit numbers as inputs and generate a 16-bit (plus carry) sum. Or you might build a circuit to multiply two numbers. These are the kinds of operations a computer’s processing unit should be able to do. Another task might be to compare two numbers to see which is larger, or to compare a set of inputs with the desired input to make sure that “all systems are go.” Or you might want to compute a “parity bit” and attach it to a number to make the

¹¹ Frank Gray, of the Bell Telephone Laboratories, received a patent for *Pulse Code Modulation* in 1953. It disclosed a “novel code” for A/D conversion, which he called a “reflected binary code,” but known thereafter as a Gray code. These were evidently discovered earlier, however, by none other than Emile Baudot (from which “baud” derives), who used them in telegraphy in 1878.

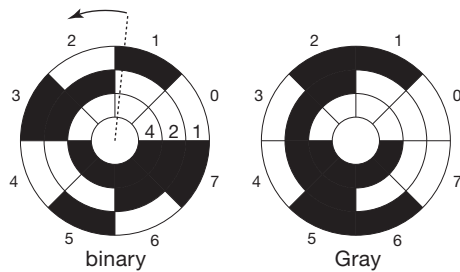


Figure 10.3. Two versions of a 3-bit angle encoder. Black or white sectors represent 1 or 0 for each of the three encoded bits (of value 4, 2, and 1 for the binary encoding). Binary encoding is prone to errors at transitions like 1→2 (shown), where more than one bit must change simultaneously; Gray code circumvents this problem.

total number of 1s even, say, before transmission over a data link; then the parity could be checked on receipt as a simple check of correct transmission. Another typical task is to take some numbers expressed in binary and display or print them as decimal characters. All of these are tasks in which the output or outputs are predetermined functions of the input or inputs. As a class, they are known as *combinational*¹² tasks. They can all be accomplished with devices called *gates*, which perform the operations of Boolean algebra applied to two-state (binary) systems.

There is a second class of problems that cannot be solved by forming a combinational function of the inputs alone, but require knowledge of past inputs as well. Their solution requires the use of *sequential* networks. Typical tasks of this type might be converting a string of bits in serial form (one after another) into a parallel set of bits, or keeping count of the number of 1s in a sequence, or recognizing a certain pattern in a sequence, or giving one output pulse for each four input pulses, or controlling the state of a system as time goes on. All these tasks require digital memory of some sort. The basic device here is the “flip-flop” (the fancy name is “bistable multivibrator”).

We begin with gates and combinational logic, since they’re basic to everything. Digital life will become more interesting when we get to sequential devices, but there will be no lack of fun and games with gates alone.

B. OR gate

The output of an OR gate is HIGH if either input (or both) is HIGH. This can be expressed in a *truth table*, as shown in Figure 10.4. The gate illustrated is a 2-input OR gate. In general, gates can have any number of inputs, but, when packaged in the form of “standard-logic” ICs, you get from

one to four gates in a single IC package.¹³ For instance, a 4-input OR gate will have a HIGH output if any one input (or more) is HIGH.

The Boolean symbol for OR is +. “A OR B” is written as $A + B$ (in text), or as $A \vee B$ (in the coding languages Verilog or C).

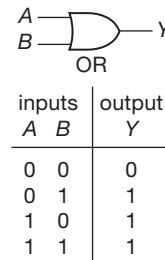


Figure 10.4. 2-input OR gate and truth table.

C. AND gate

The output of an AND gate is HIGH only if both inputs are HIGH. The logic symbol and truth table are as shown in Figure 10.5. As with OR gates, AND gates are available with 3 or 4 (sometimes more) inputs. For instance, an 8-input AND gate will have a HIGH output only if *all* inputs are HIGH.

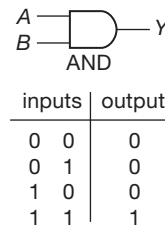


Figure 10.5. 2-input AND gate and truth table.

The Boolean symbol for AND is a dot ($\cdot$); this can be omitted, and usually is. “A AND B” is written $A \cdot B$, or simply AB (in text), or as $A \& B$ (in Verilog or C).

D. Inverter (the NOT function)

Frequently we need the complement of a logic level. That is the function of an inverter, a “gate” with only one input (Figure 10.6).

The Boolean symbol for NOT is a bar over the symbol,

¹² Sometimes called *combinatorial*.

¹³ The alternative to standard logic is “programmable logic,” in which you can connect up your own network of gates, up to hundreds of thousands within a single inexpensive IC. We’ll have plenty to say about this later in the chapter, beginning at §10.5.4, and in detail in Chapter 11.

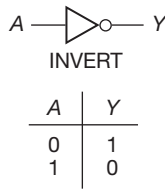


Figure 10.6. Inverter (NOT gate) and truth table.

or sometimes a prime (') symbol. “NOT A” is written $\bar{A}$ or A' . For the convenience of typesetters, the symbols /, *, −, and ' are often used in place of the overbar, to indicate NOT; thus, “NOT A” might be written as any of the following: A' , $-A$, $*A$, $/A$, $A*$, $A/$. A given document will usually pick one of these alternatives and stick with it throughout. We have chosen the form A' for this book. In coding languages NOT is written ! or ~.

An aside: propagation time

In the real world, logic devices like gates and inverters do not operate instantaneously when presented with a changed input level: it takes a *propagation time* (t_p) for the news to get from the input to the output. Figure 10.7 shows actual 'scope traces for inverter outputs from five logic families, when driven with a 15 ns active-LOW pulse. The newer low-voltage families (74AUC, 74AVLC) are fastest, with propagation times of 2 ns or less.

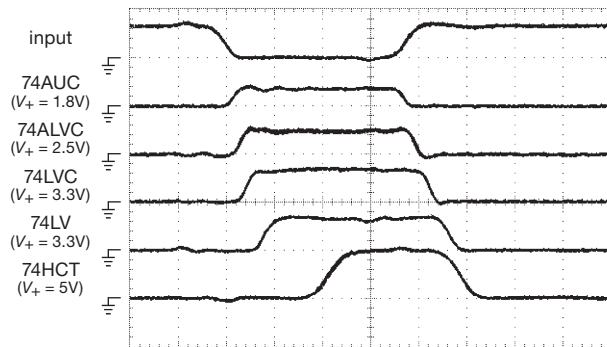


Figure 10.7. Real logic gates take a few nanoseconds to respond to a changing input. We drove inverters from five popular logic families with the active-LOW “input” pulse, and observed the outputs shown. The slowest family (74HCT, driven with a 5V swing) exhibits delays of 9 ns and 5 ns from leading and trailing edges, respectively, resulting in a shortened output pulse width. Vertical: 5 V/div; Horizontal: 4 ns/div.

E. NAND and NOR

The INVERT function can be combined with gates, forming NAND and NOR (Figure 10.8), which are somewhat more

popular than AND and OR (because, having an inversion, they can be morphed into any of the other gates, as we’ll soon see).

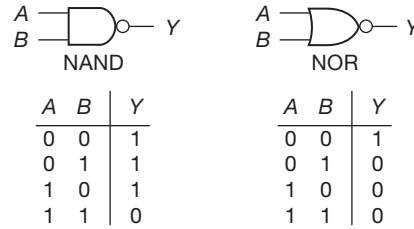


Figure 10.8. NAND and NOR, and truth tables.

F. Exclusive-OR

Exclusive-OR (XOR) is an interesting function, although less fundamental than AND and OR (Figure 10.9). The output of an XOR gate is HIGH if one or the other (but not both) input is HIGH (it never¹⁴ has more than two inputs). Another way to say it is that the output is HIGH if the inputs are different. The XOR gate is identical with modulo-2 addition of two bits. The Boolean symbol for XOR is $\oplus$. “A XOR B” is written $A \wedge B$ (in Verilog or C).¹⁵

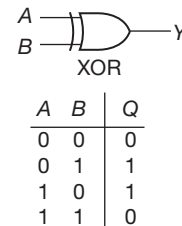


Figure 10.9. XOR and truth table.

Exercise 10.5. Show how to use the exclusive-OR gate as an “optional inverter,” i.e., it inverts an input signal or buffers it without inversion, depending on the level at a control input.

Exercise 10.6. Verify that the circuits in Figure 10.10 convert binary code to Gray code, and vice versa.

¹⁴ Well, hardly ever: the '1G386 claims to be a “3-input XOR gate,” though we would call it a 3-input parity generator.

¹⁵ Where, however, the ugly symbol $\wedge$ is used, probably because it happens to be included on standard keyboards, there intended as a foreign language accent, but evidently hijacked by the playful authors of programming languages. In the text we use the nicer-looking “wedge” symbol $\wedge$, ignoring the derisive howls of C-language purists.

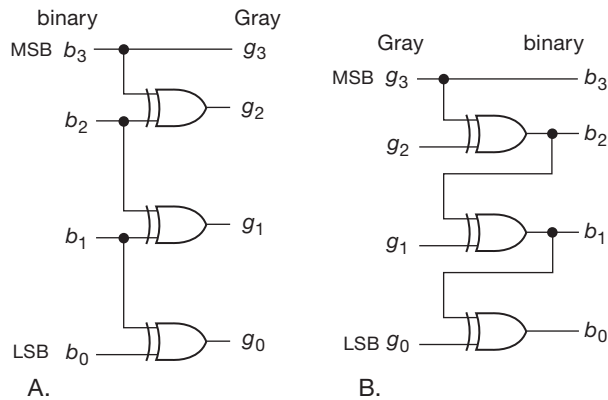


Figure 10.10. Parallel code converters: binary to Gray and Gray to binary.

	AND	OR	NOT	XOR
In text	$A B$	$A + B$	$\bar{A}$	$A \oplus B$
ABEL	$A \& B$	$A \# B$	$!A$	$A \$ B$
Verilog	$A \& B$	$A B$	$\sim A$	$A \wedge B$
VHDL	$A \text{ and } B$	$A \text{ or } B$	$\text{not } A$	$A \text{ xor } B$

Figure 10.11. Syntax for logical operations, as expressed in hardware description languages or in running text. For these operators CUPL shares the same symbols as ABEL. Alternative symbols for use in text are the following: for AND $A \cdot B$; for NOT any of the following: A' , A^* , $*A$, $/A$, $A/$, $\neg A$.

G. Basic logic: hardware description language

Thus far we've been using *schematic symbols* for the basic logic of gates. However, when you use *programmable* logic devices (PLDs: Chapter 11), rather than prefabricated standard logic, you have to enter, as text, the logic functions that you want to implement. This is done in a *hardware description language* (HDL), with names like Verilog or VHDL. Dedicated software then swallows and converts these expressions, creating a file that is used to program the actual part (or, for large volume applications, to create a “full custom” IC). The nomenclature shown in Figure 10.11 shows how these basic logical operations are expressed in the several programming languages used for programmable logic devices and custom ICs.

As a simple example, the Gray-to-binary logic shown schematically in Figure 10.10B could be written in the Ver-

ilog HDL (following some boring declarations) as:

```
assign b3 = g3;
assign b2 = g2 ^ g3;
assign b1 = g1 ^ (g2 ^ g3);
assign b0 = g0 ^ (g1 ^ (g2 ^ g3));
```

10.1.5 Discrete circuits for gates

Before going on to discuss gate applications, let's see how to make gates from discrete components. Figure 10.12 shows a diode AND gate. If either input is held LOW, the output is LOW. The output can go HIGH only when both inputs go HIGH. This circuit does work, but it has many disadvantages. In particular: (a) its LOW output is a diode drop above the signal holding the input LOW – obviously you couldn't use very many of these in a row! (b) there is no “fan-out” (the ability of one output to drive several inputs), since any load at the output is seen by the signal at the input; (c) it is slow, because of resistive pullup.

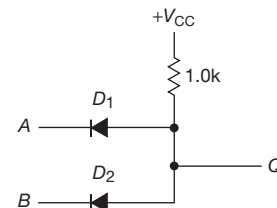


Figure 10.12. Diode AND gate.

Figure 10.13 shows how to fix some of these drawbacks, by using a pair of *npn* transistor switches to make a NOR gate, followed by an inverter to make it into an OR gate.¹⁶ A HIGH at either input (or both) turns on at least one of the input transistors, pulling their common output LOW. Since this portion of the gate is intrinsically inverting (it's a NOR), you add an inverter, as shown, to make it into a (noninverting) 2-input OR gate.

Logic circuits using bipolar transistors have been almost entirely superseded by MOS circuits. Figure 10.14 shows the analogous NOR/OR gate circuit, with *n*-channel MOS transistor switches replacing the bipolar *npn* switches of Figure 10.13. The MOS implementation has the advantage of not requiring input current (though the input capacitance means that you do have to supply current during input transitions). But it still has some drawbacks, such as limited switching speed and significant power dissipation (due to

¹⁶ This circuit was used in the family of logic known as RTL (resistor-transistor logic), which was popular in the 1960s because of its low price, but is now thoroughly obsolete.

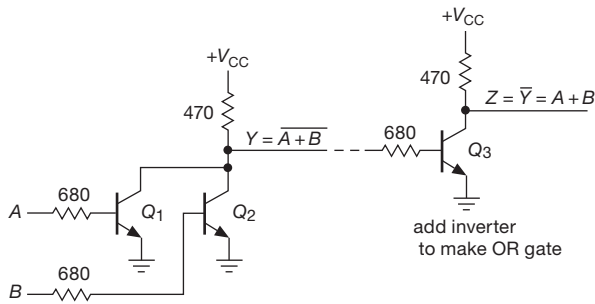


Figure 10.13. Resistor–transistor logic: NOR input stage, followed by inverter, makes a 2-input OR gate.

resistive pullup resistors). Both problems are elegantly finessed by the use of complementary MOS (“CMOS”) transistors in a push–pull arrangement, as we discussed earlier in §3.4.4A.

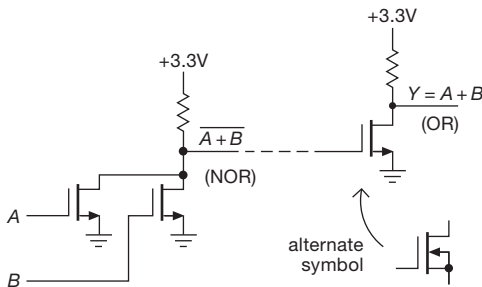


Figure 10.14. An nMOS OR gate. Digital circuit designers simplify the MOSFET symbol, always omitting the substrate terminal; they usually center the gate attachment, too, and indicate polarity instead with a bubble on the gate, as shown in the bottom pair of Figure 3.6.

Although the discrete gate circuits just illustrated are simple to understand, you wouldn’t use them in practice because of their disadvantages. In fact, except in rare circumstances¹⁷ you would never construct gates (or any other logic) from discrete components, since a full range of excellent logic is available as inexpensive and compact ICs, as we will see shortly. Currently nearly all IC logic circuits are built with complementary MOSFETs (CMOS). Look

¹⁷ In some industrial and consumer applications, where ruggedness and simplicity are important, you’ll find circuits like Figure 10.13, constructed from discrete components. For this purpose you can get transistors with integrated base resistors (or base divider); these are called “bias resistor transistors” (BRTs), “pre-biased transistors,” or, sometimes, “digital transistors.” They’re ridiculously inexpensive, as little as \$0.02 apiece in quantity. They’re made by companies like ON semiconductor, Diodes Inc., and Rohm.

back at Figure 3.91 to remind yourself how you would make a CMOS NAND gate.

10.1.6 Gate-logic example

Let’s work out a circuit to perform the logic we gave as an example in Chapters 1 and 2: the task to sound a buzzer if either car door is open and the driver is seated. The answer is obvious if you restate the problem as “output HIGH if either the left door OR the right door is open, AND driver is seated,” i.e., $Q = (L + R)S$. Figure 10.15 shows it with gates. The output of the OR gate is HIGH if one OR the other door (or both) is open. If that is so, AND the driver is seated, Q goes HIGH. With an additional transistor, this could be made to sound a buzzer or close a relay.¹⁸

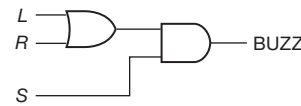


Figure 10.15. Car-door example: active-HIGH levels.

In practice, the switches generating the inputs will probably close a circuit to ground, to save extra wiring (among other reasons). This means, for example, that the inputs will go LOW when a door is opened. In other words, we have “active-LOW” inputs. Let’s rework the example with this in mind, calling the inputs L' , R' , and S' .

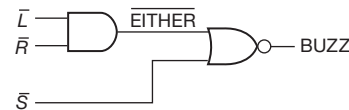


Figure 10.16. Car-door example: active-LOW levels cause confusion.

First, we need to know if either door input (L' , R') is LOW; i.e., we must distinguish the state “both inputs HIGH” from all others. That’s an AND gate. So we make L' and R' the inputs to an AND gate. The output will be LOW if either input is LOW; call that $EITHER'$. Now we need to know when $EITHER'$ is LOW and S' is LOW; i.e., we must distinguish the state “both inputs LOW” from all others. That’s an OR gate. Figure 10.16 shows the circuit. We have used a NOR gate, instead of an OR gate, to get the same output as

¹⁸ This particular gate circuit just happens to be made, in a tiny minilogic package, as a 74LVC1G3208; they call it a “3-input positive OR–AND gate.” But don’t expect this level of service for all your gate circuit needs.

earlier, namely the Q output is HIGH when the desired condition is present. Something strange seems to be going on here, though. We have used AND instead of OR (and vice versa), as compared with the earlier circuit. Section 10.1.7 should clarify the matter. First, though, consider the following exercise.

Exercise 10.7. What do the circuits shown in Figure 10.17 do?

A. Gate interchangeability

When designing digital circuits, keep in mind that it is possible to form one kind of gate from another. For example, if you need an AND gate, and you have half of a 74LVC00 available (quad 2-input NAND), you can substitute as shown in Figure 10.18. The second NAND functions as an inverter, making AND. The following exercises should help you explore this idea.

Exercise 10.8. Using 2-input gates, show how to make (a) INVERT from NOR, (b) OR from NORs, and (c) OR from NANDs.

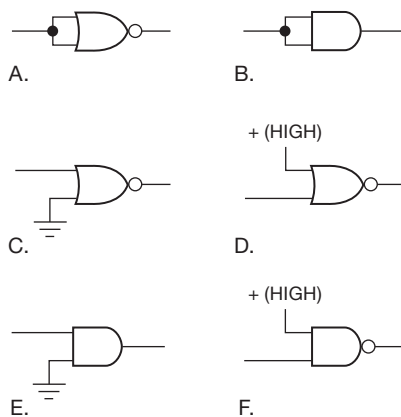


Figure 10.17. Gate configurations for Exercise 10.7.



Figure 10.18. Making AND from NANDs.

Exercise 10.9. Show how to make (a) a 3-input AND from 2-input ANDs, (b) a 3-input OR from 2-input ORs, (c) a 3-input NOR from 2-input NORs, and (d) a 3-input AND from 2-input NANDs.

In general, multiple use of one kind of inverting gate (e.g., NAND) is enough to make any combinational function. However, this isn't true for a noninverting gate, because there's no way to make INVERT. This probably ac-

counts for the greater popularity of NAND and NOR in logic design.

10.1.7 Assertion-level logic notation

An AND gate has a HIGH output if both inputs are HIGH. So if HIGH means "true," you get a true output only if all inputs are true. In other words, with active-HIGH logic, an AND gate performs the AND function. The same holds for OR.

What happens if LOW means "true," as in the last example? An AND gate gives a LOW if either input is true (LOW): it's an OR function! Similarly an OR gate gives a LOW only if both inputs are true (LOW). It's an AND function! Very confusing.

There are two ways to handle this problem. The first way is to think through any digital design problem as we did earlier, choosing the kind of gate that gives the needed output. For instance, if you need to know if any of three inputs is LOW, use a 3-input NAND gate. This method is still used by some misguided designers. When designing this way, you would draw a NAND gate, even though the gate is performing a NOR function on the (active-LOW) inputs. You would probably label the inputs as in Figure 10.19. In this example, $CLEAR'$, MR' (master reset), and $RESET'$ might be active-LOW levels coming from various places in a circuit. The output, CLR, is active-HIGH and will go to the devices that are to be cleared if *any* of the reset signals goes LOW (true).

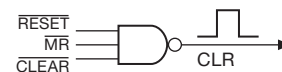


Figure 10.19. Confusing notation for OR function of active-LOW signals.

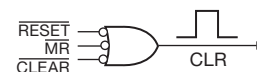


Figure 10.20. Use inverted input "bubbles" for active-LOW inputs.

The second way to handle the problem of active-LOW signals is to use "assertion-level logic." If a gate performs an OR function on active-LOW inputs, draw it that way, as in Figure 10.20. The 3-input OR gate with negated inputs is functionally identical with the preceding 3-input NAND. That equivalence turns out to be an important logical identity, as stated in DeMorgan's theorem, and we will spell out a number of such useful identities shortly. For now, it is enough to know that you can change AND to OR (and

vice versa) if you negate the output and all inputs (see Table 10.3). Assertion-level logic looks forbidding at first, because of the proliferation of funny-looking gates. It is better, though, because the logical functions of the gates in the circuit stand out clearly. You'll find it friendly after you've used it for a while, and you won't want to use anything else.

Let's rework the car-door example again with assertion-level logic (Figure 10.21). The gate on the left determines if L or R is true, i.e., LOW, giving an active-LOW output. The second gate gives a HIGH output if both $(L + R)$ and S are true, i.e., LOW. From DeMorgan's theorem (after a while you won't even need that, you'll recognize these gates as equivalent) the first gate is AND and the second is NOR, just as in the circuit drawn earlier. Two important points.

1. Active-LOW (or LOW-true) is sometimes called "negative-true," but that doesn't mean that the logic levels are of negative *polarity*.¹⁹ It means that the lower of the two states (LOW) stands for TRUE.
2. The symbol used to draw the gate itself assumes active-HIGH logic. A NAND gate used as an OR for active-LOW signals could be drawn as a NAND, or (better) using assertion-level logic, as an OR with negation symbols (bubbles) at the inputs. In the latter case you think of the bubbles as indicating inversion of the input signals, followed by an OR gate operating on active-HIGH input logic levels as originally defined.²⁰

You might well ask why you shouldn't just keep things simple by doing all your designs with active-HIGH logic. In some cases you are constrained by the logic levels defined by the components themselves (for example, the common use of an active-LOW reset input on a microcontroller); and in other situations (like the car-door switches) it is better electrically to connect the common terminal to ground. In any case, you've got to be able to navigate in a digital world populated by both active-HIGH and active-LOW signals.

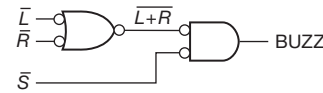


Figure 10.21. Car-door redux: assertion-level logic fixes confusing Figure 10.16.

10.2 Digital integrated circuits: CMOS and Bipolar (TTL)

Digital logic functions are implemented in hardware ICs, either as (small-scale) *standard logic* (e.g., the 74xx gates we've seen), or as *programmable logic* (e.g., an FPGA – field-programmable gate array; §11.2.3), or as a full-custom application-specific IC (ASIC, or ASSP²¹; e.g., a graphics processor). Because this book is intended primarily for the *circuit* designer (as opposed to the *chip* designer²²), we will not discuss the design of ICs themselves.

CMOS dominates contemporary digital IC technology, having largely replaced the earlier bipolar ("TTL") logic. CMOS is faster, better adapted to operation at low supply voltage, and less power hungry. There are numerous families within CMOS (and also within bipolar); they offer the same logic functions, the differences having to do primarily with speed, supply voltage, and output drive capability. There are more than a half-dozen manufacturers of digital logic, with much duplication ("second sourcing"). For example, you can get a quad 2-input NAND in the popular LVC family (its part number is 74LVC00) from five manufacturers, and the largely similar 74LCX00 from three other manufacturers.

We discuss these family choices in some detail in §10.2.2 and §12.1.1. As a preview, though, and to get a sense of the march of technology, take a look at Figure 10.22, which shows the life cycle of most of the important digital logic families. Bipolar logic's days are numbered (except for BiCMOS – CMOS logic with bipolar output – and also some specialty families such as the speedy ECL).

Without going into detail now (see §12.1.1), we would suggest, for most purposes, either the mature HC(T) family, or the more recent (and faster) LVC/LVX family. The former is widely available, has an enormous variety of logic functions, and includes through-hole (dual in-line package or DIP) as well as surface-mount technology (SMT) packaging; the latter is faster, optimized for performance at lower supply voltages, but available in SMT only.

²¹ Application-Specific Standard Product.

²² Sometimes you hear the terms *board-level* design and *chip-level* design.

¹⁹ A confusion that can lead to real damage: we loaned a digital frequency synthesizer to an inexperienced student, who read the manual and then applied -5 V to the programming inputs. He spent many days replacing burned-out circuitry.

²⁰ Logical AND and OR shouldn't be confused with the *legal* equivalents. The weighty legal tome known as *Words and Phrases* has over 40 pages of situations in which AND can be construed as OR. Among the more amusing is this gem: "OR will be construed AND, and AND will be construed OR, as the necessities of the case may require. . . ." This isn't the same as DeMorgan's theorem!

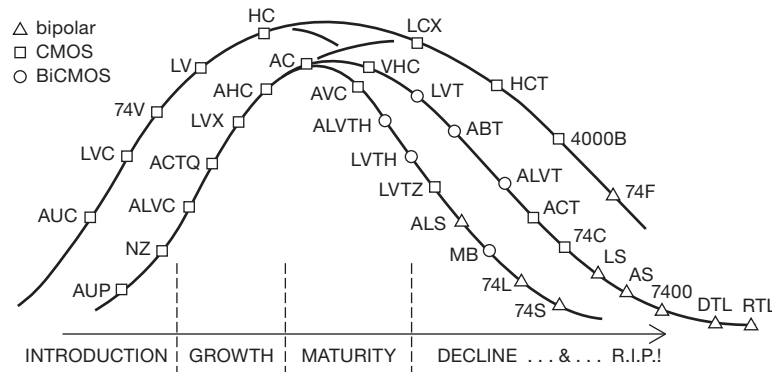


Figure 10.22. Life cycle of logic families, at a snapshot in time early in the third millennium. CMOS is in the ascendancy, bipolar in decline. We put this figure together by looking at analogous figures from NXP, TI, and others; by checking stock at distributors; and by factoring in our own biases.

10.2.1 Catalog of common gates

Table 10.3 shows the common gates you can get as standard digital logic. Each gate is drawn in its normal (active-HIGH inputs) incarnation, and also the way it looks for active-LOW inputs. These functions come in traditional 14- or 16-pin packages, with multiple gates per package (limited by total pin count); they are also available singly in tiny packages.²³ Figure 10.23 shows what these packages look like, going from the traditional through-hole DIP down to the tiny grain-of-sand CSP (chip-scale package) packages on the lower right; you could fit the latter parts nicely between any two leads of the former!

To fully specify one of these gates, you would form a part number beginning with 74, then add the several letters (like LVC, for low-voltage CMOS) to specify the family, then the numbers that designate the function (like 08, for 2-input AND). Additionally you would add some suffixes to specify package and temperature range, and perhaps a prefix like “SN” to specify the manufacturer: voilà, an SN74LVC08ADR, which is a quad 2-input AND in the LVC family, in reels of 2500 parts, packaged in a 14-pin “SOIC” (small-outline IC) temperature range of -40°C to $+125^{\circ}\text{C}$, manufactured by Texas Instruments. For simplicity (and prevention of insanity) we routinely omit most of this detail, indicating digital IC types with an apostrophe,

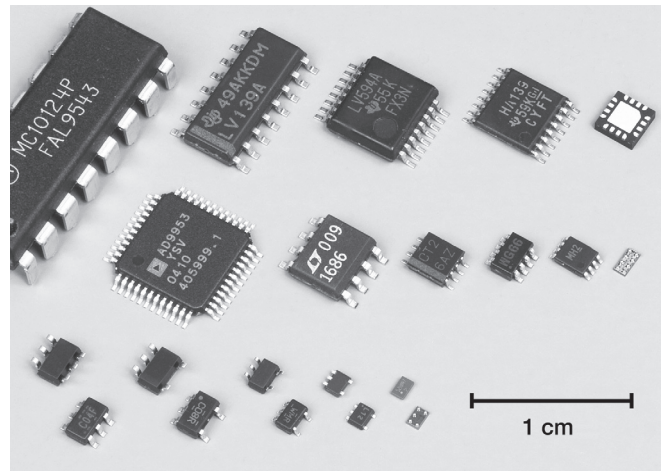


Figure 10.23. A selection of digital logic packages; all are surface mount, except for the DIP-16 cameo at upper left. Top row, left to right: DIP-16, SOIC-16, SSOP-16, TSSOP-16, QFN-16. Middle row: TQFP-48, SOIC-8, SSOP-8, SOT-23-8, US-8, WCSP-8 (DSBGA-8: die-size ball-grid array). Bottom row (two samples each): SOT-23-6, SOT-23-5, SC-70, SOT-533, WCSP-5 (DSBGA-5).

e.g., '08 for a 2-input AND function; in situations where the family type matters, we add the family designation, e.g., 'LVC08.

A. A “universal” gate?

Figure 10.24 shows an elegant trick, namely a tiny 6-pin part (called a '1G97) whose small block of logic lets you make any of nine logic functions, according to how you connect the inputs: inverter, noninverting buffer, 2-input multiplexer (MUX, see §10.3.3A), and six varieties of

²³ In a curious sort of inversion, the part numbers are shortest for the large parts, and vice versa. Here's an example: a 2-input NAND gate, packaged as a quad (4 gates) in a 14-pin package, has part number SN74LVC00DR; the same function, packaged singly in a 5-pin package, is called an SN74LVC1G00MDBVREP. (The “1G” portion specifies what we'll call “minilogic” – the packaging of single logic units in small-pin-count packages.)

Table 10.3 Standard Logic Gates in Popular Families^a

Function	Symbol		Designator ^b	Inputs	Output ^c	AUC	AUP	ALVC	LVC	LCX	LVX	LV	AHC(T)	VHC(T)	AC(T)(Q)	HC(T)	F	LS	7N, 7S, 7W	4000B		100E, EL, EP	
	Active-H inputs	Active-L inputs																					
AND			'08	2	PP	1-4	1	4	1-4	4	4	4	1-4	1-4	4	4	4	4	1-2	4	1-4-5	4	1-4-5
			'11	3	PP	-	1-3	-	1	3	-	3	-	3	3	3	3	3	1	3	-	3	-
			'21	4	PP	-	-	-	-	-	-	2	-	2	-	2	2	2	-	2	-	2	-
NAND			'00, '37	2	PP	1-2	1-2	4	1-2-4	4	4	4	1-2-4	1-4	4	4	4	4	1-2	4	1-4-5	4	1-4-5
			'03, '38	2	OD	-	1-2	-	1-2	4	-	-	-	-	-	-	-	-	1-2	-	-	-	-
			'10	3	PP	-	1	-	1	-	-	3	-	-	-	-	-	-	1	3	-	3	-
			'20	4	PP	-	-	-	-	-	-	2	-	-	2	2	2	2	-	2	-	2	-
			'30	8	PP	-	-	-	-	-	-	-	-	-	-	1	1	1	-	1	-	1	-
OR			'32	2	PP	1-4	1	4	1-2-3-4	4	4	4	1-4	1-4	4	4	4	4	1-2	4	-	4	-
			'332	3	PP	-	1	-	1	-	-	-	-	-	-	-	-	-	1	-	-	-	-
			'802	4	PP	-	-	-	-	-	-	-	-	-	-	2	-	-	-	2	1-4	2	1-4
NOR			'02	2	PP	1-2-4	1-2	-	1-2-4	4	4	4	1-4	1-4	4	4	4	4	1-2	4	-	4	-
			'27	3	PP	-	1	-	1	3	-	3	-	3	-	3	3	3	1	3	-	3	-
			'25	4	PP	-	-	-	-	-	-	-	-	2	-	-	-	-	-	2	1-4	2	1-4
INVERT (NOT)			'04		PP	1-2-6	1-2-3	6	1-2-3-6	6	6	6	1-6	1-6	6	6	6	6	1-2-3	6	-	6	-
			'14 ($\overline{\text{J}}$)		PP	1-2	1-2	-	1-2-3-6	6	6	-	-	1-6	-	-	-	-	1-2-3	-	-	-	-
			'240		3S	1-2	1-2	-	1	6	-	-	-	6	-	-	-	-	2	-	-	-	-
			'05, '06		OD	1-2-6	1-2-3	-	1-2-3-6	6	-	6	6	1-6	6	6	-	6	1	-	-	-	-
BUFFER			'34		PP	2-6	1-2	-	1	-	-	-	-	-	-	-	-	-	1-2-3	6	-	6	-
			'125, '126		3S	1	1	6	1	6	6	-	1-6	1-6	-	6	6	6	1-2	6	-	6	-
			'07, '17		OD	1-2	1-2	-	1-2	6	-	-	1	1	-	-	-	-	1-2-3	-	-	-	-
			'241, '244, '541		3S	6	-	6	6	6	6	6	6	6	6	6	6	6	-	-	-	-	-
XOR			'86	2	PP	1-2	-	-	1-2-4	4	4	4	1-4	1-4	4	4	4	4	1-2	4	1-5	4	1-5
			'386	3	PP	-	-	-	1	-	-	-	-	-	-	-	-	-	1	-	-	-	-
UNIVERSAL			'57-8, '97-8	3	PP	-	1	-	1	-	-	-	-	-	-	-	-	-	1	-	-	-	-
			'99	4	3S	-	1	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Notes: (a) available gates/pkg listed as $m \cdot n$ etc.; e.g., a 2-input OR ('32) in the AUC family is available with one or four gates in a package. (b) the digits signifying logic function follow the family designator (e.g., 2-input AND gates: 74LVC08, 74LVC1G08); common to all families except HV CMOS (4000B) and ECL (100E, EL, EP). (c) PP=push-pull (active pull-up & pull-down); OD=open-drain; 3S=3-state.

2-input gates (AND, OR, AND with one inverted input, OR with one inverted input, NAND with one inverted input, or NOR with one inverted input). The companion '1G98 (same logic, but with inverted output) also has nine disguises, three of which are different (namely NAND, NOR, and MUX with inverted output). The smallest package for these parts is a mere 0.9×1.4 mm, way too small a space to

print its part number.²⁴ Going one step further, the '1G99 squanders two extra pins to give you (a) selectable output inversion (via an XOR) and (b) a "three-state" output

²⁴ Try it: a full part number is "SN74AUP1G97YZPR," and this $\diamond$ is the package size (actually it's a bit generous, but we couldn't find a small enough symbol in the $\text{\LaTeX}$ typesetting language).

(coming up in §10.2.4A). The similar '1G57 and '1G58 gates have Schmitt-trigger inputs.

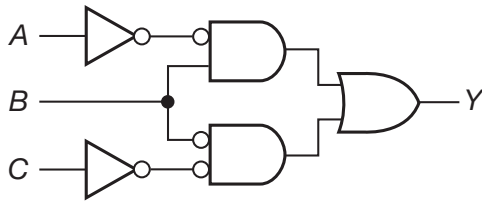


Figure 10.24. The '1G97 and '1G98 “configurable multiple-function gates” can perform any of nine logic functions. The '1G97 is shown; the '1G98 is identical, except with an inverted output.

Exercise 10.10. Show how to connect up a '1G97 to make each of the nine functions just listed.

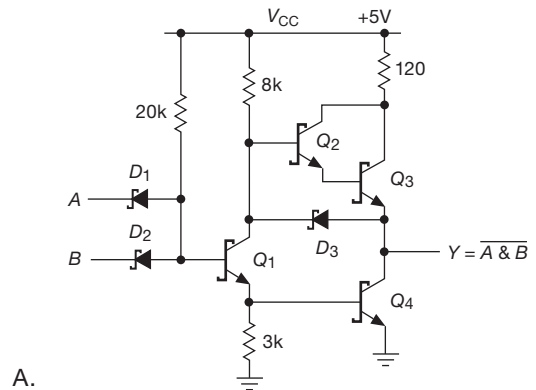
10.2.2 IC gate circuits

Although a NAND gate, for instance, performs identical logic operations in the various family versions, the logic levels and other characteristics (speed, power, input current, etc.) are quite different. In general, you have to be careful when mixing logic-family types. To understand the differences, look at the schematics of a NAND gate in Figure 10.25.

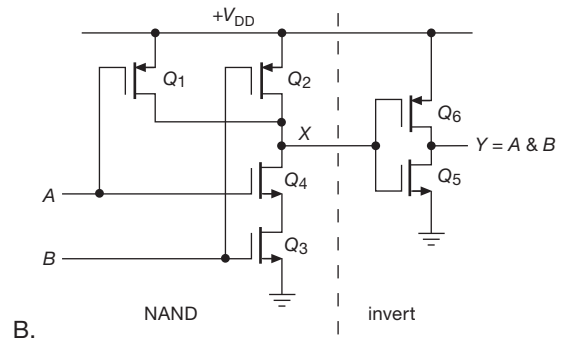
The CMOS gate (by far the most common family type) is constructed from enhancement-mode MOSFETs of both polarities, connected as switches (rather than followers). An ON FET looks like a low resistance (R_{on}) to whichever supply rail it is connected. Both inputs must be HIGH to turn on the series pair Q_3Q_4 and to turn off both of the pullup transistors Q_1Q_2 . That produces a LOW at the output (marked X), i.e., it is a NAND gate. Q_5 and Q_6 constitute the standard CMOS inverter, thus completing the AND gate. From this example it should be evident how to generalize to AND, NAND, OR, and NOR with any number of inputs.

Exercise 10.11. Draw the circuit of a 3-input CMOS OR gate.

Bipolar-transistor logic families are no longer preferred because they are outperformed by CMOS families;²⁵ but it is instructive to look at the legacy “TTL” family.²⁶ The once-popular LS (low-power Schottky) NAND gate (Figure 10.25A consists basically of the diode-resistor logic



A.



B.

Figure 10.25. NAND/AND gates: two circuit implementations. A. Bipolar LS TTL NAND gate, with its “totem-pole” output stage; B. CMOS AND gate.

of Figure 10.12 driving a transistor inverter followed by a push-pull output. If both inputs are HIGH, the 20k resistor holds Q_1 on, thus producing a LOW output by saturating Q_4 and shutting off Darlington Q_2Q_3 . If at least one input is LOW, Q_1 is held off, thus producing a HIGH output by follower action of Q_2Q_3 combined with Q_4 being held off. (Note that the HIGH output, coming from a Darlington follower, is at least two diode drops below the +5 V supply.) Schottky diodes and Schottky-clamped transistors are used throughout for enhanced speed.²⁷

Note that both CMOS and bipolar TTL gates have an output circuit with “active pullup” to the positive supply rail, unlike our discrete gate examples (Figures 10.12–10.14).

²⁵ With the exception of the extremely fast ECL types and the hybrid “BiCMOS” hefty bus-driver families.

²⁶ Among other reasons, its ghost hovers over much contemporary logic, in the form of “TTL-input level” limits of ≤ 0.8 V (LOW) and ≥ 2.0 V (HIGH), seen for example in several logic families in Figure 10.2.

²⁷ Schottky diodes have no stored charge and therefore no reverse recovery delay (see, e.g., §9.5.3B or §9x.6); and Schottky clamping prevents transistor saturation, which otherwise causes a turn-off delay.

10.2.3 CMOS and bipolar (“TTL”) characteristics

This section might be subtitled “Analog aspects of digital circuits.” The same logic function (e.g., NAND) can be implemented in different ways; these can differ in their *electrical* characteristics, while performing the same *logic*. In a nutshell – **Supply voltage**: CMOS can operate over a range, TTL requires +5 V; **Input current**: CMOS inputs draw no steady current, TTL inputs require current; **Input voltage**: the various families have varying logic thresholds and allowable input voltage, therefore incompatibilities; **Output**: CMOS outputs are rail-to-rail, TTL cannot reach V_+ ; **Speed and power**: CMOS has only *dynamic* power consumption (proportional to frequency), whereas TTL has substantial quiescent power; and the fastest families are low-voltage CMOS and bipolar ECL.

In somewhat greater detail (cracking open the nutshell):

Supply voltage: In Figure 10.26 we’ve plotted the supply-voltage ranges for most of the interesting logic families. The various CMOS families each have a healthy range of allowable supply voltages; for example, the LVC family is fully specified for supply voltages from +1.8 V to +3.3 V, and most family members support operation to 5 V.²⁸ Within its range a CMOS device runs faster at higher supply voltages (where there’s more gate drive voltage). The bipolar families run on a single voltage: +5 V $\pm 5\%$ for TTL, and either –5 V (sometimes –5.2 V) or +5 V for ECL (called NECL and PECL, for negative and positive ECL, respectively).

Input – Current: CMOS devices have no quiescent input current (apart from leakage); however, like all devices, their input capacitance (of order 4 pF) draws current during switching ($I = C dV/dt$; so, for example, a 2.5 V input transition in 2 ns would require ~ 5 mA of drive current). Bipolar logic does require quiescent input current: a TTL input held in the LOW state sources current into whatever drives it (e.g., 0.6 mA typ for the F family), so to hold it LOW you must sink current (in addition to the capacitive load current during switching).²⁹ In general, logic families have adequate output-current capability to drive additional logic; what matters more is compatibility of logic-level *voltages*.

– **Logic Levels**: CMOS families generally put their input threshold voltage at half the supply voltage (though

with considerable spread, typically 1/3 to 2/3 of the supply voltage); this is a good choice, since CMOS outputs swing all the way to both rails. However, for compatibility with legacy bipolar TTL output levels (where the HIGH output is well below the positive supply), many CMOS families have “TTL threshold” variants, often specified with a “T” in the family name: HC→HCT, VHC→VHCT. These families specify a maximum LOW threshold of +0.8 V and a minimum HIGH threshold of +2.0 V (see Figure 10.2, and the more complete Figure 12.6). These duplicate the bipolar-TTL specification, where the input logic threshold is about two diode drops above ground (about 1.3V).³⁰

– **Voltage Tolerance**: The world has not standardized on a single logic supply voltage (nor should it), so in a typical digital system you’ll usually have several supply voltages (e.g., +5 V and +3.3 V). Hence the question: can the output of logic running at one supply voltage (call it X) drive the input of logic at a different supply voltage (call it Y)? The short answer (we’ll have a longer answer in Chapter 12) is that two things are required: (a) X’s output levels have to satisfy Y’s input logic-level requirements; and (b) if Y’s supply voltage is less than X’s, Y’s inputs must tolerate X’s (larger) output voltages. This latter is called input-voltage tolerance, and you’ve got to respect it! For example, you can see in Figure 10.2 that the legacy HC(T) family will not tolerate inputs greater than its supply voltage,³¹ whereas the newer LVC family accepts inputs to +5.5 V regardless of its own supply voltage (including when it is unpowered). Input voltage tolerance is essential when digital signals cross over supply-voltage boundaries.

CMOS inputs are susceptible to damage from static electricity during handling. Unused inputs should be tied HIGH or LOW, as necessary (more on this in §10.8.3B).

Output: CMOS outputs are driven by a pair of MOSFET switches, either to ground or to V_+ ; i.e., “rail-to-rail.” The TTL output stage, by contrast, is a saturated transistor to ground in the LOW state and a (Darlington) follower in the HIGH state (two diode drops below V_+). The datasheet will usually give more details, specifying typical and worst-case output voltages for some typical load currents.³²

²⁸ For most LVC logic devices the datasheets say the “recommended” operating range extends only to +3.6 V, but some LVC parts extend that to +5.5 V, with their operating characteristics specified at 5 V.

²⁹ The niche bipolar ECL is strange stuff – its outputs are “bare emitters,” which are deliberately terminated with 50 Ω resistors to –2 V (NECL) or +3 V (PECL).

³⁰ This bit of history has cast its shadow all over the digital world, with “TTL input levels” pretty much established as standard for nearly all devices with digital inputs.

³¹ More precisely, the inputs may not go more than 0.5 V above V_+ or below ground.

³² CMOS devices usually specify output voltage at symmetrical pairs of

In general, the faster families (ALVC, LVC, LCX; F, AS) have greater output drive capability than the slower families (CD4000, HC(T); LS).

Speed and power: All CMOS families of standard logic consume zero quiescent current.³³ However, their power consumption rises linearly with increasing frequency, because switching the capacitance of internal nodes and external capacitive loads requires current ($I = C dV/dt$). CMOS operated near its upper frequency limit may even dissipate more power than bipolar logic (see Figure 10.27). It’s common to see this *dynamic current*³⁴ specified in terms of an effective “power-dissipation capacitance,” C_{pd} , from which you can calculate the no-load dynamic power dissipation as $P_{diss} = C_{pd} V^2 f$ (there are two transitions per cycle, which cancels the usual factor of 1/2). For example, a 74LVC00 (quad NAND gate) specifies $C_{pd} = 19$ pF per gate, from which you get a power dissipation of 0.2 mW/MHz per gate (for 3.3 V supply); so one such IC with all four gates churning along at 100 MHz would dissipate 80 mW internally (and additional power from switching the external load capacitances). The speed range of CMOS standard-logic functions goes from about 2 MHz (for the high-voltage CD4000 series running at a low 5 V) to about 100 MHz (for AHCT/VHCT at 5 V) to about 150 MHz (for LVC/LCX at 3.3 V) to about 350 MHz (for AUC at 2.5 V). In contrast to CMOS’s zero quiescent current, the bipolar TTL families consume considerable quiescent current, more for the faster families (AS, F, ABT); the corresponding speeds go from about 25 MHz (for LS) to about 100 MHz (for AS and F).

In Figure 10.26 (seen again as Figure 12.3, with additional commentary) we’ve plotted worst-case gate propagation delays for commonly used standard logic families.

In general, the nice characteristics of the CMOS families (zero quiescent current, rail-to-rail output swings,

HIGH- and LOW-state load currents, for example at ± 8 mA; TTL devices, with their unsymmetrical output, are usually specified at plenty of sink current, but very stingy source currents (e.g., at 8 mA and at -0.4 mA). This is important when a logic output is used to drive some external component, e.g., an LED indicator or solid-state relay: wire the component so the logic output *sinks* current (with the other end returned to the positive supply, through a current-limiting resistor if necessary).

³³ Large-scale CMOS circuits (for example gate arrays or microprocessors, as opposed to the basic “standard-logic” functions like gates and flip-flops) generally have nonzero (and sometimes quite substantial) quiescent current.

³⁴ When charging and discharging a capacitor from 0 to V at frequency f , the average current is $I = fCV$.

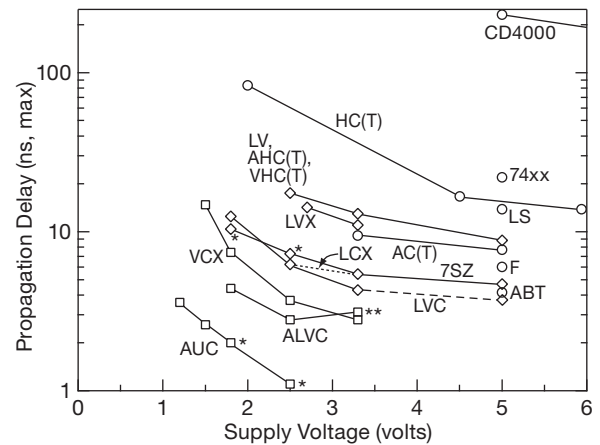


Figure 10.26. Gate speed versus supply voltage, for popular logic families. Maximum specified propagation delay ($t_{pd(max)}$) is shown for the standard supply voltages at which each family is specified. (As a rough guide, “typical” delays are in the range of 35–75% of $t_{pd(max)}$.) See the caption to Figure 12.3 for details.

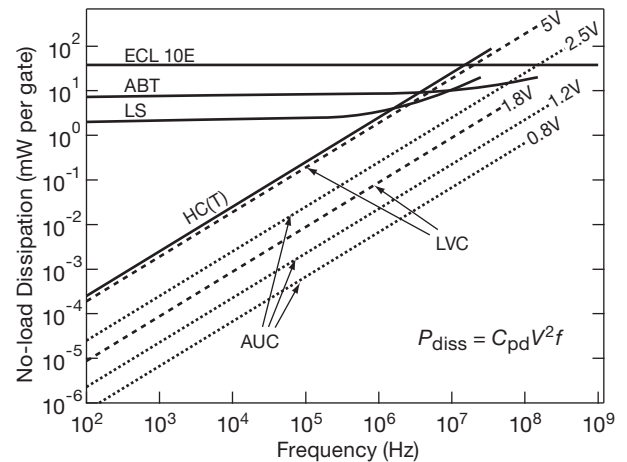


Figure 10.27. Gate internal power dissipation versus frequency for some CMOS and bipolar logic families. Note the dramatic (quadratic) dependence on power supply voltage, within any one family. See also Figure 12.2.

symmetrical sink–source output currents, and high speed) make them the logic of choice. Among them, the LVC and 7SZ families are nice,³⁵ with their 5 V tolerant inputs and good supply voltage ranges (1.8–3.3 V³⁶ and 1.8–5 V). For operation at 5 V the AHC(T), VHC(T), or LV families are

³⁵ The 7SZ and 17SZ families come only in minilogic packages.

³⁶ The maximum “recommended” supply voltage for some LVC devices is 5.5 V, whereas for others it is only 3.6 V; however, while the latter types give no specifications for operation above 3.6 V, *all* LVC devices grudgingly allow supply voltages to 5.5 V.

also good choices; their inputs are 5 V tolerant regardless of supply voltage (which can be 2.5–5 V). These parts are available only in surface-mount packages; if you want DIP through-hole parts for ease of breadboarding, use HC(T) or AC(T).

For unusual applications you might choose CD4000B series parts (supply voltages to 15 V, but *slow!*), ECL logic (*fast!* – to 1 GHz), or the hybrid (BiCMOS) ABT series (output current to 64 mA, good for driving heavy loads such as buses).

Within any one logic family, outputs are designed to drive other inputs easily, so you don't often have to worry about thresholds, input current, etc. For instance, with bipolar TTL, any output can drive at least 10 other inputs (the official term for this is *fan-out*: TTL has a fan-out of 10), so you don't have to do anything special to ensure compatibility. In Chapter 12 we go into the issue of interfacing between logic families and between logic circuits and the outside world.

10.2.4 Three-state and open-collector devices

The CMOS and TTL gates we have just discussed have push–pull output circuits: the output is held either HIGH or LOW by an ON transistor. Nearly all digital logic uses this sort of circuit (called active pullup; in TTL it's also known as a totem-pole output) because it provides low output impedance in both states, giving faster switching time and better noise immunity, as compared with an alternative such as a single transistor with a passive collector-pullup resistor. It also results in lower power dissipation.

However, there are a few situations for which active pullup output is unsuitable. As an example, imagine a computer system in which several functional units have to exchange data. The central processor unit (CPU), memory, and various peripherals all need to be able to send and receive 16-bit words. It would be awkward (to put it mildly) to have separate 16-wire cables connecting each device to all others. The solution is the so-called *data bus*, a single set of 16 wires accessible to all devices. It's like the old-fashioned (now extinct?³⁷) telephone “party line”: only one device at a time may “talk” (assert data), but all may “listen” (receive data). With a bus system there must be agreement as to who may talk, and you'll see words like *bus arbitration* and *bus master*.

You can't use gates (or any other devices) with active pullup–pulldown outputs to drive a bus, since you couldn't disconnect your output from the shared data lines (you're

holding it either HIGH or LOW at all times). What's needed is a gate whose output can be *open*. Such devices are available, and they come in two varieties, *three-state* devices and *open-collector* devices.

A. Three-state logic

Three-state logic, also called TRI-STATE[®] logic (a trademark of National Semiconductor Corporation, NSC), provides an elegant solution. The name is misleading; it is not digital logic with three voltage levels. It's just ordinary logic, with a third output state: open circuit (Figure 10.28). A separate *enable* input determines whether the output behaves like an ordinary active pullup output or goes into the “third” (open) state, regardless of the logic levels present at the other inputs. Three-state outputs are available on many digital chips, including counters, latches, registers, etc., as well as on gates and inverters. A device with three-state output behaves exactly like ordinary active pullup logic when enabled, always driving its output either HIGH or LOW; when disabled,³⁸ it effectively disconnects its output, so another logic device can drive the same line. Let's look at an example.

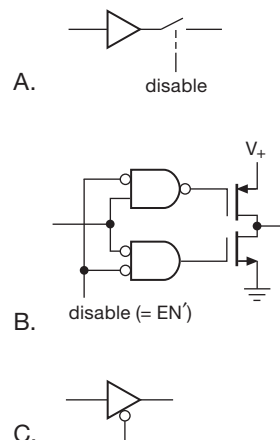


Figure 10.28. Three-state CMOS logic; A. Conceptual diagram. B. Realization with internal CMOS gates. C. Logic symbol.

B. A look ahead: data buses

Three-state drivers are widely used to drive computer data buses. Every device (memory, peripherals, etc.) that needs to put data on the (shared) bus ties onto it with three-state gates (or more complex functions such as “registers”). Things are cleverly arranged so that at most one device has its drivers enabled at any instant, all other devices being

³⁷ But playing a major role in the 1959 movie *Pillow Talk*.

³⁸ Mr. Lebowsky is *disabled*, yes.

disabled into the open (third) state. In a typical situation the selected device “knows” to assert data onto the bus by recognizing its particular address on a set of address and control lines (Figure 10.29). In this simplified case the device is wired as port 6: it looks at address lines A_0 – A_2 and asserts data onto data bus D_0 – D_3 when it sees its particular address (i.e., 6) on the address lines *and* it sees a READ pulse. Such a bus protocol is adequate for many simple systems. Something like this is used in most microcomputers, as you will see in Chapter 14.

Note that there must be some external logic to make sure that three-state devices sharing the same output lines don’t try to talk at the same time (that undesirable condition is officially called “bus contention”). In this case all is well as long as each device responds to a unique address.

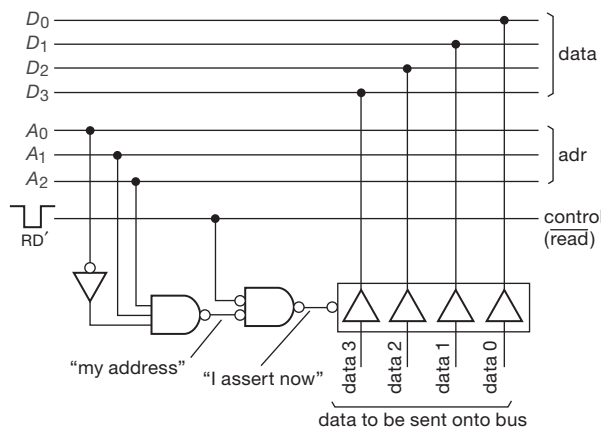


Figure 10.29. Data bus with address decode logic and three-state drivers.

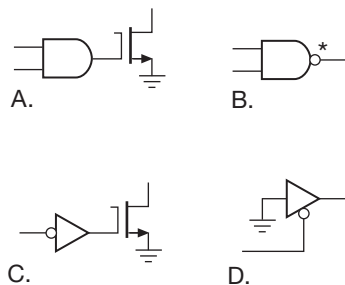


Figure 10.30. Open-drain logic: A. open-drain NAND; B. symbol; C. open-drain noninverting buffer; D. implementation with three-state buffer.

C. Open-collector and open-drain logic

The predecessor to three-state logic was “open-collector” logic, which allows you to share a single line among the

outputs of several drivers. An open-collector (or open-drain) output simply omits the active pullup transistor of the output stage (Figure 10.30). The name “open-collector” is a good one. When you use such gates, you must supply an external pullup resistor somewhere. Its value isn’t critical; a small-value resistor gives increased speed and improved noise immunity, at the expense of increased power dissipation and loading of the driver. Values of a few hundred to a few thousand ohms are typical. If you wanted to drive a bus with open-collector gates (rather than with three-state drivers), you would substitute 2-input open-collector NANDs for the three-state drivers of Figure 10.29, bringing one input of each gate HIGH to enable the gates onto the bus; note that the data bits then asserted onto the bus are inverted. Each bus line would need a single resistive pullup to the positive supply.

The disadvantage of open-collector logic is that speed and noise immunity are degraded, when compared with logic constructed with active pullup devices, because of the resistive pullup circuit. That’s why three-state drivers are nearly universally favored for computer bus applications. However, there are three situations in which you would choose open-collector (or open-drain) devices: driving external loads, “wired-OR,” and external buses. Let’s look at them briefly.

D. Driving external loads

Open-collector (O/C) logic is good for driving external loads that are returned to a higher-voltage positive supply. You might want to drive a low-current lamp or relay that requires 12 V, or perhaps just generate a 15 V logic swing by running a resistor from a gate’s output to +15 V, as in Figure 10.31. A popular O/C device is the ULN2003/4, a seven-channel open collector Darlington array with internal clamp diodes (for inductive loads); it accepts direct logic drive, has a 50 V breakdown rating, and can switch up to 500 mA (the 75468/9 is similar, but with 100 V breakdown). More on these subjects in §12.4.

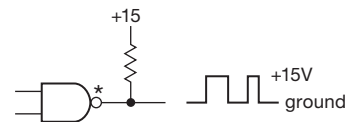


Figure 10.31. Open-collector logic as level translator.

E. Wired-OR

If you wire together some open-collector gates as shown in Figure 10.32, you get what’s called “wired-OR” – the combination behaves in this case like a larger NOR gate,

with the output going LOW if any input is HIGH. You can't do this with active pullup outputs, because there would be a contest of wills if all the gates didn't agree on what the output should be. You can combine NORs, NANDs, etc., with this kind of connection, and the output will be LOW if any gate asserts a LOW output. This connection is sometimes called "wired-AND," because the output is HIGH only if all gates have HIGH (open) outputs. Both names are describing the same thing: it's wired-AND for active-HIGH logic and wired-OR for active-LOW logic. This will make more sense to you after you've seen DeMorgan's theorem in the next section.

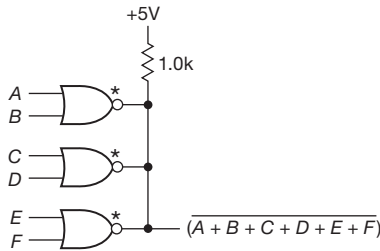


Figure 10.32. Wired-OR.

Wired-OR enjoyed some brief popularity in the early days of digital electronics, but it is not much used today, with two exceptions: (a) in the family of logic known as ECL (emitter-coupled logic) the outputs are what you might call "open-emitter," and can be wired-OR'ed painlessly; and (b) there are some shared lines in computer buses (most notably the line called *interrupt*) whose function is not to transfer data bits, but merely to indicate if *at least one* device is requesting attention; in that case you use wired-OR, because it does what you want and doesn't require external logic to prevent contention.

F. External buses

Where speed is not too important, you sometimes see open-collector drivers used to drive buses. Examples are the original SCSI bus used to connect disks and peripherals, and the IEEE-488 (also called general-purpose interface bus, GPIB) instrument bus. More on this in Chapter 14.

10.3 Combinational logic

As we discussed earlier in §10.1.4A, digital logic can be divided into *combinational* (sometimes called *combinatorial*) and *sequential*. Combinational circuits are those in which the output state depends on only the present input states in some predetermined fashion, whereas in sequential circuits the output state depends both on the input states

and the previous history. Combinational circuits can be constructed with gates alone, whereas sequential circuits require some form of memory (flip-flops). In these subsections we explore the possibilities of combinational logic before entering the turbulent world of sequential circuits.

Table 10.4 Logic Identities

$ABC = (AB)C = A(BC)$
$AB = BA$
$AA = A$
$A1 = A$
$A0 = 0$
$A(B + C) = AB + AC$
$A + AB = A$
$A + BC = (A + B)(A + C)$
$A + B + C = (A + B) + C = A + (B + C)$
$A + B = B + A$
$A + A = A$
$A + 1 = 1$
$A + 0 = A$
$1' = 0$
$0' = 1$
$A + A' = 1$
$AA' = 0$
$(A')' = A$
$A + A'B = A + B$
$(A + B)' = A'B'$
$(AB)' = A' + B'$

10.3.1 Logic identities

No discussion of combinational logic is complete without the identities shown in Table 10.4. Most of these are obvious. The last two comprise DeMorgan's theorem, the most important for circuit design.

A. Example: exclusive-OR gate

We illustrate the use of the identities with an example: making the exclusive-OR function from ordinary gates. Figure 10.33 shows the XOR truth table. From studying this and by realizing that the output is 1 only when $(A, B) = (0, 1)$ or $(1, 0)$, we can write

$$A \oplus B = \bar{A}B + A\bar{B},$$

from which we have the realization shown in Figure 10.34. However, this realization is not unique. Applying the identities, we find

$$\begin{aligned}
 A \oplus B &= A\bar{A} + A\bar{B} + B\bar{A} + B\bar{B} \\
 &\quad (A\bar{A} = B\bar{B} = 0) \\
 &= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\
 &= A(\overline{AB}) + B(\overline{AB}) \\
 &= (A + B)(\overline{AB}).
 \end{aligned}$$

(In the first step we used the trick of adding two quantities that equal zero; in the third step we used DeMorgan's theorem.) This has the realization shown in Figure 10.35. There are still other ways to construct XOR. Consider the following exercise.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 10.33. XOR truth table.

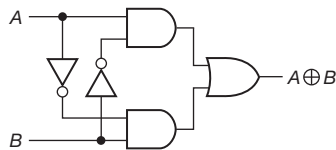


Figure 10.34. XOR realization.

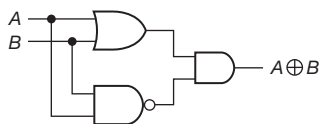


Figure 10.35. Another XOR realization.

Exercise 10.12. Show that

$$\begin{aligned}
 A \oplus B &= \overline{AB + \bar{A}\bar{B}}, \\
 A \oplus B &= (A + B)(\bar{A} + \bar{B}),
 \end{aligned}$$

by logic manipulation. You should be able to convince yourself that these are true by inspection of the truth table, combined with suitable hand-waving.

Exercise 10.13. What are the following: (a) $0 \cdot 1$, (b) $0 + 1$, (c) $1 \cdot 1$, (d) $1 + 1$, (e) $A(A + B)$, (f) $A(A' + B)$, (g) $A \text{ XOR } A$, (h) $A \text{ XOR } A'$?

10.3.2 Minimization and Karnaugh maps

Because a realization of a logic function (even one as simple as exclusive-OR) isn't unique, it is often desirable to find the simplest, or perhaps most conveniently constructed, circuit for a given function. Many good minds have worked on this problem, and there are several methods available, including algebraic techniques widely available as software. For example, all of the "hardware description languages" (HDLs) used to enter circuitry going into programmable logic (see §11.2.6) include automatic logic minimization; you don't even see it happening.

Perhaps of historical interest only, you'll sometimes hear the term *Karnaugh map*, which is a simple tabular method for minimizing logic with four or fewer inputs; it also enables you to find a logic expression (if you don't know it) once you can write down the truth table.

We illustrate the method with an example (and then abandon it entirely!). Suppose you want to generate a logic circuit to count votes. Imagine that you have three active-HIGH inputs (each either 1 or 0) and an output (0 or 1). The output is to be 1 if at least two of the inputs are 1.

Step 1. Make a truth table:

A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

All possible permutations must be represented, with corresponding output(s). Write an X (= "don't care") if either output state is OK.

Step 2. Make a Karnaugh map. This is somewhat akin to a truth table, but the variables are represented along two axes. Furthermore, they are arranged in such a way that only one input bit changes in going from one square to an adjacent square (Figure 10.36).

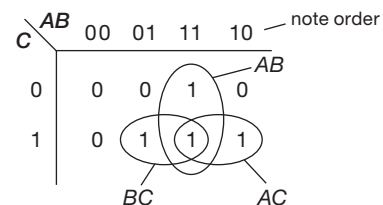


Figure 10.36. Karnaugh map for majority logic.

Step 3. Identify on the map groups of 1s (alternatively, you could use groups of 0s): the three blobs (called a “cover”) enclose the logic expressions AB , AC , and BC . Finally, read off the required function, in this case

$$Q = AB + AC + BC$$

with the realization shown in Figure 10.37. The result seems obvious, in retrospect. We could have read off the pattern of 0s to get instead

$$Q' = A'B' + A'C' + B'C',$$

which might be useful if the complements A' , B' , and C' already exist somewhere in the circuit.

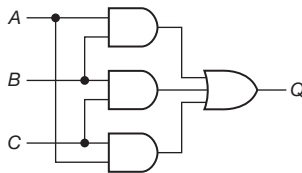


Figure 10.37. Majority (voting) logic.

Exercise 10.14. Just for fun, draw a Karnaugh map for logic to determine if a 3-bit integer (0 to 7) is prime (assume that 0, 1, and 2 are not primes). Show a realization with 2-input gates.

Exercise 10.15. Continuing Karnaugh-map fun: find logic to perform multiplication of two 2-bit unsigned numbers (i.e., each 0 to 3), producing a 4-bit result. *Hint:* use a separate Karnaugh map for each output bit.

10.3.3 Combinational functions available as ICs

With logic gates alone, you could construct logic to perform rather complicated functions such as binary addition or magnitude comparison, parity checking, multiplexing (selecting one of several inputs, as determined by a binary address), etc. In fact, that is just what is done when you implement complex logic in *gate arrays*³⁹ or in other forms of *programmable logic* (see §10.3.3F, §10.5.4, and Chapter 11.). Programmable logic (usually combined with a microcontroller) is quite often the method of choice when you want to make some digital (or combined analog/digital) system. We’ll illustrate these techniques in the next chapter.

However, these functions are generally available also in the form of prefabricated MSI chips (medium-scale integration, upward of 100 gates on one chip), to be used as

standard-logic functions. Although many interesting MSI functions involve flip-flops (i.e., *sequential* circuits, which we will get to shortly), there are a number of them that are combinational functions involving gates only. Let’s see what animals live in the MSI combinational zoo.

A. 2-input select (multiplexer)

The 2-input select (also called a 2-input *multiplexer*, or “MUX”) is a very useful function. It is basically a two-position switch for logic signals. Figure 10.38 shows the basic idea, with both a discrete gate implementation and an IC that packages four 2-input multiplexers (a “quad MUX”) in one IC. When SELECT is LOW, the A inputs are passed through to their respective Y outputs; when SELECT is HIGH, the B inputs appear at the output. Holding ENABLE’ HIGH disables the device by forcing all outputs LOW. This is an important concept we’ll see more of later. Here’s the truth table, which illustrates the X (don’t care) entry:

Inputs				Outputs
E'	SEL	A_n	B_n	Y_n
H	X	X	X	L
L	L	L	X	L
L	L	H	X	H
L	H	X	L	L
L	H	X	H	H

It can be written more compactly like this:

Inputs		Outputs
E'	SEL	Y_n
H	X	L
L	L	A_n
L	H	B_n

In a hardware description language the 2-input MUX logic function (without an ENABLE input) would be written as $Y = \sim S \& A \mid S \& B$; with an ENABLE input it would become $Y = E \& (\sim S \& A \mid S \& B)$.⁴⁰ Figure 10.38 and the preceding table correspond to the ’157 quad 2-input select chip. The same function is also available with inverted output (’158) and with three-state outputs (true: ’257; inverted: ’258). It is also available as a tiny single-section MUX (without the EN' input), with part numbers ’1G157 and ’2G157. These chips work electrically like gates – they do the logic and regenerate a logic level at the output accordingly. Another way to implement this function is with a few *transmission gates*, in which the appropriate input signal is simply passed through to the

³⁹ A function so implemented is sometimes amusingly referred to as having been “Gatorade.”

⁴⁰ For a somewhat obscure reason (having to do with a “logic-race” condition) it’s a good idea to add the redundant term $A \& B$ to these expressions; thus $Y = E \& (\sim S \& A \mid S \& B \mid A \& B)$.

output (via MOS transistors) without regeneration; we'll see this next.

Exercise 10.16. Show how to make a 2-input select, using a pair of 3-state buffers and whatever other logic you need.

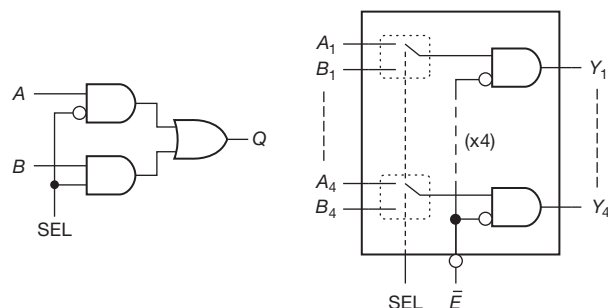


Figure 10.38. Two-input “select” gates: A. implemented with discrete gates; B. packaged together as a “quad,” with a single SELECT line.

Although the function of a select gate can be performed by a mechanical switch in some cases, the gate is a far better solution, for several reasons:

- (a) it is cheaper;
- (b) all channels are switched simultaneously and rapidly;
- (c) it can be switched, nearly instantaneously, by a logic level generated elsewhere in the circuit (most interestingly, from a microprocessor or other smart device);
- (d) even if the select function is to be controlled by a front-panel switch, it is better not to run fast logic signals around through cables and switches, to avoid capacitive signal degradation and noise pickup.

With a select gate actuated by a dc level, you keep logic signals on the circuit board and get the bonus of simpler off-board wiring, namely a single line with pullup switched to ground by a single-pole single-throw (SPST) switch. Controlling circuit functions with externally generated dc levels in this manner is known as “cold switching,” and it is a much better approach than controlling the signals themselves with switches, potentiometers, etc. Besides its other advantages, cold switching lets you bypass control lines with capacitors to eliminate interference, whereas signal lines cannot generally be bypassed. We'll see some examples of cold switching later.

B. Transmission gates

As we discussed in §3.4.1A, with CMOS it is possible to make “transmission gates,” simply a pair of complementary MOSFET switches in parallel, so that an input (ana-

log) signal between ground and V_{DD} is either connected through to the output through a low resistance (less than a hundred ohms) or open-circuited (essentially infinite resistance). As you may remember, such a device is bidirectional and doesn't know (or care) which end is input and which end is output. Transmission gates work perfectly well with digital CMOS levels, and in fact they are used extensively in the internal circuitry of CMOS digital devices. You can get them also as standard logic ICs. Figure 10.39 shows the layout of the popular '4066 CMOS “quad bilateral switch.” Each switch has a separate control input: HIGH closes the switch, and LOW opens it. They are available also in compact single-section and two-section packages ('1G66, '2G66). Note that transmission gates are merely switches, and therefore have no fan-out; i.e., they simply pass input logic levels through to the output without providing additional drive capability.⁴¹

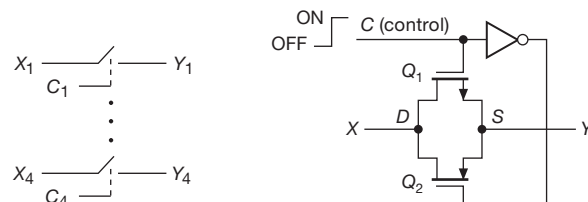


Figure 10.39. Quad transmission gate; a single gate, implemented with MOS transistors, is shown at right.

With transmission gates you can make 2-input (or more) select functions, usable with CMOS digital levels or analog signals. To select among a number of inputs, you can use a bunch of transmission gates (generating the control signals with a *decoder*, as will be explained later). This is such a useful logic function that it has been institutionalized as the multiplexer, discussed next.

Exercise 10.17. Show how to make a 2-input select with transmission gates. You will need an inverter.

C. Multiplexers with many inputs

Multiplexers are available with 4, 8, and 16 inputs. A binary address is used to select which of the input signals appears at the output. For instance, an 8-input MUX has a 3-bit address input to address the selected data input (Figure 10.40). The digital MUX illustrated is a '151. It has an active-LOW STROBE' (another name for ENABLE') input, and it provides true and complemented outputs. When the

⁴¹ Transmission gates are attractive to IC designers because their simple design requires little area on the silicon die, and they do not incur the switching delay of a conventional gate.

```

wire [1:0] A; // 2 address ("select") input lines
wire [3:0] D; // 4 input data lines
wire Y, YBAR, ENBAR // outputs (true and complemented), and enable
assign Y = ~ENBAR & ( D[0] & ~A[1] & ~A[0]
                    | D[1] & ~A[1] & A[0]
                    | D[2] & A[1] & ~A[0]
                    | D[3] & A[1] & A[0] );
assign YBAR = ~ Y;

```

Figure 10.41. Verilog code for a 4-input multiplexer.

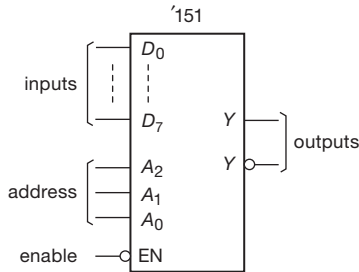


Figure 10.40. '151 8-input multiplexer.

chip is disabled (ENABLE' held HIGH), Y is LOW and Y' is HIGH, independent of the states of the address and data inputs.

It's a good idea to become somewhat familiar with description languages used for logic entry, if for no reason other than anxiety reduction. Figure 10.41 shows what Verilog code looks like, for the simpler example of a 4-input multiplexer.

Electrically, two varieties of multiplexers are available. One type is for digital levels only, with an input threshold and "clean" regeneration of output levels according to the input state: an example is the '153 logic MUX (available in both CMOS and bipolar logic families). The other kind of MUX is analog and bidirectional; it's really just an array of transmission gates. These come only in CMOS families, and can be used for both logic and analog signals. The '4051–'4053 CMOS multiplexers work this way. Remember that logic made from transmission gates has no fan-out. Since transmission gates are bidirectional, these multiplexers can be used as "demultiplexers," or decoders. We discuss them next.

Exercise 10.18. Show how to make a 4-input multiplexer using (a) ordinary gates, (b) gates with three-state outputs, and (c) transmission gates. Under what circumstances would (c) be preferable?

You might wonder what to do if you want to select among more inputs than are provided in a multiplexer. This

question comes under the general category of chip "expansion" (using several chips that have small individual capabilities to generate a larger capability), and it applies to decoders, memories, shift registers, arithmetic logic, and many other functions as well. In this case the job is easy (Figure 10.42). Here we have expanded two 74LS151 8-input multiplexers into a 16-input multiplexer (note that we've used lowercase designations for the input and output signals, to prevent confusion with the chip's similarly named pins).⁴² There's an additional address bit, of course, and you use it to enable one chip or the other. The disabled chip holds its Y LOW, so an OR gate at the output completes the expansion. With three-state outputs the job is even simpler, because you can connect the outputs directly together.

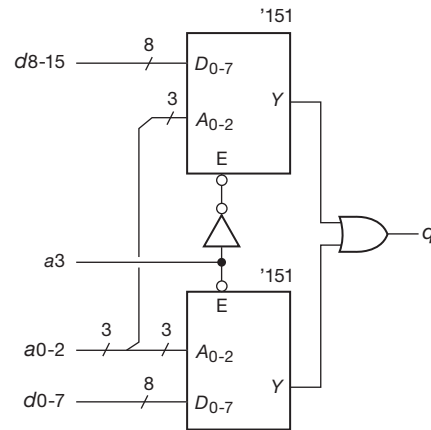


Figure 10.42. Multiplexer expansion.

D. Demultiplexers and decoders

A demultiplexer is the opposite of a multiplexer: it takes an input and routes it to one of several possible outputs,

⁴² We've also introduced a common "bus" convention, namely the use of a single line with a diagonal slash to indicate a collection of similar signal lines; the number by the slash tells you how many signals are in the group, and the label tells you what they are.

```

wire [1:0] A; // 2-bit address: brackets show array range
wire [3:0] YBAR; // four active-LOW outputs
wire ENBAR; // active-LOW enable
assign YBAR[0] = ~(~ENBAR & ~A[1] & ~A[0]);
assign YBAR[1] = ~(~ENBAR & ~A[1] & A[0]);
assign YBAR[2] = ~(~ENBAR & A[1] & ~A[0]);
assign YBAR[3] = ~(~ENBAR & A[1] & A[0]);

```

Figure 10.44. Verilog code for one section of a '139 dual 1-of-4 decoder.

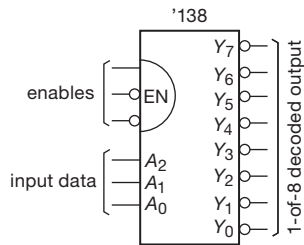


Figure 10.43. '138 1-of-8 decoder.

according to an input binary address. The other outputs are either held in the inactive state or open-circuited, depending on the type of demultiplexer.

A decoder is similar, except that the address is the only input, and it is “decoded” to assert one of n possible outputs. Figure 10.43 shows an example. This is the '138 “1-of-8 decoder.” The output corresponding to (addressed by) the 3-bit input data is LOW; all others are HIGH. This particular decoder has three ENABLE inputs, all of which must be asserted (two LOW, one HIGH); otherwise all outputs are HIGH.

Decoders are commonly used when interfacing to a data bus, to trigger different actions depending on the address; we'll treat this subject in detail in Chapter 13. Another common use of a decoder is to enable a sequence of actions in turn, according to an advancing address given by the output of a binary *counter* (§10.4.2E). A close cousin of the '138 is the '139, a dual 1-of-4 decoder with a single active-LOW ENABLE per section. Figure 10.44 illustrates what its Verilog code would look like.

Figure 10.45 shows how to use a pair of '138 1-of-8 decoders to generate a 1-of-16 decoder. No external gates are necessary, since the '138 has ENABLE inputs of both polarities.

Exercise 10.19. More expansion: make a 1-of-64 decoder from nine '138s. *Hint:* use one of them as an enabling switchyard for the others.

In CMOS logic, the multiplexers that use transmission gates are also demultiplexers, since transmission gates are

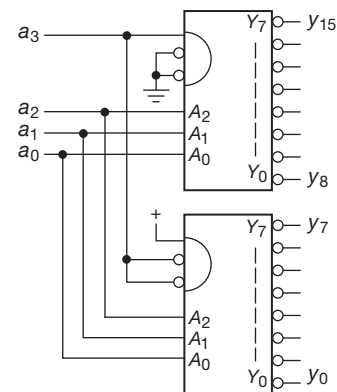


Figure 10.45. Decoder expansion.

bidirectional. When they are used that way, it is important to realize that the outputs that aren't selected are open-circuited. A pullup or pulldown resistor, or equivalent, must be used to assert a well-defined logic level on those outputs (the same requirement as with TTL open-collector gates).

Another kind of decoder is the '47 “BCD-to-7-segment decoder/driver.” It takes a BCD input and generates outputs on 7 lines corresponding to the segments of a “7-segment display” that have to be lit to display the decimal character. This type of decoder is really an example of a *code converter*, but in common usage it is called a decoder.

Exercise 10.20. Design a BCD-to-decimal (1-of-10) decoder using gates.

Exercise 10.21. Design a “simple” encoder: a circuit that outputs the (2-bit) address telling which of 4 inputs is HIGH (all other inputs must be LOW).

Exercise 10.22. Figure out how to make a parity generator with XOR gates.

E. Other arithmetic chips

The *priority encoder* generates a binary code giving the address of the highest-numbered input that is asserted. It is particularly useful in “parallel-conversion” ADCs (see

next chapter) and in microprocessor system design. An example is the '148 8-input (3 output bits) priority encoder. The '147 encodes 10 inputs.

Figure 10.46 shows a 4-bit *magnitude comparator*. It determines the relative sizes of the 4-bit input numbers A and B and tells you via outputs whether $A < B$, $A = B$, or $A > B$. Inputs are provided for expansion to numbers larger than 4 bits.

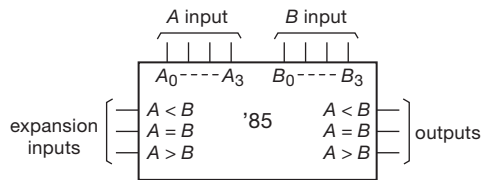


Figure 10.46. Magnitude comparator.

A *parity-generator* chip is used to generate a parity bit to be attached to a “word” when transmitting (or recording) data, and to check the received parity when such data are recovered. Parity can be even or odd (e.g., with odd parity the number of 1 bits in each character is odd). The '280 parity generator, for instance, accepts a 9-bit input word, giving an even- and an odd-parity bit output. The basic construction is an array of exclusive-OR gates.

Figure 10.47 shows a 4-bit *full adder*.⁴³ It adds the 4-bit number A_i to the 4-bit number B_i , generating a 4-bit sum S_i plus carry bit C_o . Adders can be “expanded” to add larger numbers: the “carry-in” input C_i is provided to accept the carry out of the next lower adder.

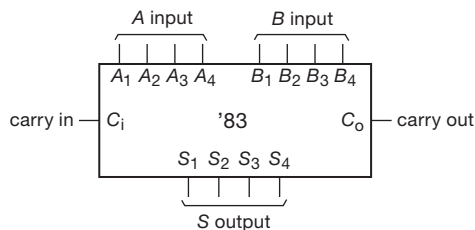


Figure 10.47. 4-bit full adder.

Figure 10.48 shows the Verilog code (here written out in somewhat longwinded form, for clarity) for a 4-bit full adder; see if you can understand how it figures out the sum.

⁴³ We show the '83 logic-family part number mostly from a sense of nostalgia. The part does not exist in the 'HC family, and the 7483 and 74LS83 parts are obsolete. In contemporary design such MSI (medium-scale integration) functions are most often implemented with some form of programmable logic (see §10.5.4 and Chapter 11). In fact, it is the very superiority of the latter that brought about the demise of many of the old MSI parts.

A nice thing about languages like Verilog or VHDL is that they understand higher levels of abstraction (and you can, too, once you get the hang of them). In this case all the stuff after the wire declarations can be written as a single line, namely `assign {COUT,S} = A + B + CIN;`.

A device known as an *arithmetic logic unit* (ALU) can be used as an adder, though it has the capability of performing a number of different functions. For instance, the '181 4-bit ALU (expandable to larger word lengths) can do addition, subtraction, bit shifts, magnitude comparison, and a few other functions. Adders and ALUs do their arithmetic in times measured in nanoseconds to tens of nanoseconds, depending on logic family. Other dedicated arithmetic chips include the *multiplier-accumulator* (MAC), which accumulates a sum of products, and the *correlator*, which compares the corresponding bits of a pair of bit strings, calculating the number of bits that agree.

However, owing to the development of large and fast microprocessors, contemporary digital design favors either general-purpose microprocessors or the more optimized digital signal processors (DSPs) for the kind of signal processing that involves extensive arithmetic functions. An attractive alternative is the FPGA, user-configurable to be just about anything; you can drop in a “soft” processor or other function, and you can get them with optimized “hard” functions already built in. We’ll talk about configurable logic in the next chapter, and microprocessors in Chapter 15.

F. Programmable logic devices

You can configure your own custom combinational (and sequential) logic on a single chip by using ICs that contain an array of gates with programmable interconnections. These are known generically as *programmable logic devices* (PLDs). The popular varieties are cPLDs (complex PLDs) and field-programmable gate arrays (FPGAs). Both flavors are high achievers, flexible, and a delight to use. They are a part of every designer’s toolbox of tricks. We’ll meet them later in this chapter, and again in the next.

10.4 Sequential logic

10.4.1 Devices with memory: flip-flops

All our work with digital logic so far has been with combinational circuits (e.g., arrays of gates), for which the output is determined completely by the existing state of the inputs. There is no “memory,” no history, in these circuits. Digital life gets really interesting when we add devices with memory. This makes it possible to construct counters, arithmetic

```

wire [3:0] A;
wire [3:0] B;
wire [3:0] S; // sum bits
wire CIN, COUT; // carries
assign S[0] = A[0] ^ B[0] ^ CIN; // recall "^" means xor
assign C01 = A[0] & B[0] | A[0] & CIN | B[0] & CIN;
assign S[1] = A[1] ^ B[1] ^ C01;
assign C12 = A[1] & B[1] | A[1] & C01 | B[1] & C01;
assign S[2] = A[2] ^ B[2] ^ C12;
assign C23 = A[2] & B[2] | A[2] & C12 | B[2] & C12;
assign S[3] = A[3] ^ B[3] ^ C23;
assign COUT = A[3] & B[3] | A[3] & C23 | B[3] & C23;

```

Figure 10.48. Verilog code for a 4-bit full adder.

accumulators, and circuits that generally do one interesting thing after another. The basic unit is the flip-flop, a colorful name to describe a device that, in its simplest form, looks as shown in Figure 10.49.

Assume that both A and B are HIGH. What are X and Y ? If X is HIGH, then both inputs of G_2 are HIGH, making Y LOW. This is consistent with X being HIGH, so we're finished. Right?

$X = \text{HIGH},$
 $Y = \text{LOW}.$

Wrong! The circuit is symmetrical, so an equally good state is

$X = \text{LOW},$
 $Y = \text{HIGH}.$

The states X, Y both LOW, and X, Y both HIGH, are not possible (remember, $A = B = \text{HIGH}$). So the flip-flop has two stable states (it's sometimes called a "bistable"). Which state it is in depends on past history. It has memory! To write into the memory, just bring one of the inputs momentarily LOW. For instance, bringing A LOW momentarily guarantees that the flip-flop goes into the state

$X = \text{HIGH},$
 $Y = \text{LOW},$

no matter what state it was in previously. You would describe it as an "SR flip-flop," which is Set or Reset with an active-LOW input pulse.

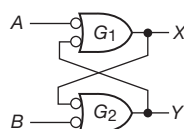


Figure 10.49. Flip-flop ("set-reset" type).

A. Switch debouncing

This kind of flip-flop (with a SET and RESET input, often labeled S and R) is quite useful in many applications. Figure 10.50 shows a typical example. This circuit is supposed to enable the gate and pass input pulses when the switch is opened.⁴⁴ The problem with this circuit is that switch contacts *bounce*. When the switch is closed, the two contacts actually separate and reconnect, typically 10 to 100 times over a period of about a millisecond. You would get the kind of waveforms shown; if there were a counter or shift register using the output, it would faithfully respond to all those extra "pulses" caused by the bounce.

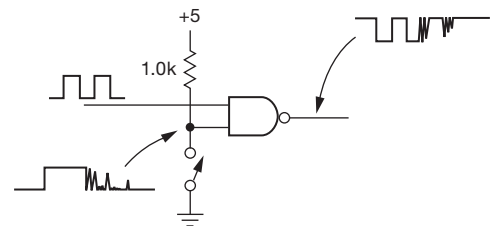


Figure 10.50. Mechanical-switch "bounce."

Figure 10.51 shows a cure. The flip-flop changes state when the contacts first close. Further bouncing against that contact makes no difference and SPDT (single-pole, double-throw) switches never bounce all the way back to the opposite position; the output is a *debounced* signal, as sketched. This debouncer circuit is widely used; the '279 "quad SR latch" lets you get four into one package, and the '1G74 (in a smaller package) can be used when only a single flip-flop is needed. Incidentally, the preceding circuit

⁴⁴ We generally like to tie a switch to ground (not +5V) for a couple of reasons: (1) ground is a convenient (and electrically "quiet") return path for switches and other controls; and (2) we got used to it, owing to a peculiarity of (current-sourcing) bipolar logic.

has a minor flaw: the first pulse after the gate is enabled may be shortened, depending on when the switch is closed relative to the input pulse train; the same holds for the final pulse of a sequence (of course, a switch that is not debounced has the same problem). A “synchronizer” circuit (see §10.4.4) can be used to prevent this from happening, for applications where it makes a difference.

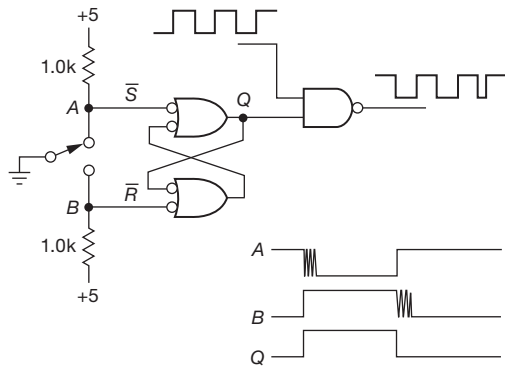


Figure 10.51. SR flip-flop switch debouncer. Note the damaged output waveform. The nodes labelled $\bar{S}$ and $\bar{R}$ are active-LOW SET and RESET inputs to the flip-flop.

Figure 10.52A shows a trick we use for making a simpler debouncer: the noninverting buffer (which could be made instead from an '08 2-input AND, or two cascaded '04 inverters, or whatever) holds its last state (just like a flip-flop). The switch, when thrown, momentarily overrides the buffer's output; the latter, however, is wise enough not to fight inevitability, and it switches (without bounce) to agree. There's a lot of output current during the momentary contention, which, however, lasts only during the gate transition time, of order a few nanoseconds. No harm done, and, despite one's feeling of discomfort at such a dirty trick,⁴⁵ it works just fine. With a pair of cascaded inverters (Figure 10.52B) you can avoid the connection to V_{DD} .

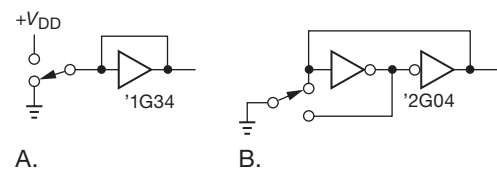


Figure 10.52. Simple switch debouncer.

⁴⁵ Timid engineers have been known to replace the feedback wire with a resistor, say 1 k Ω . This eliminates the momentary current spike on the power rail, simultaneously relieving your lingering anxiety.

B. Multiple-input flip-flop

Figure 10.53 shows another simple flip-flop. Here NOR gates are used; a HIGH input forces the corresponding output LOW. Multiple inputs allow various signals to set or clear the flip-flop. In this circuit fragment, no pullups are used, because logic signals generated elsewhere (by standard active pullup outputs) are used as inputs.

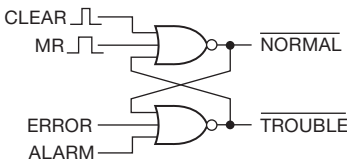


Figure 10.53. SR flip-flop with multiple inputs.

10.4.2 Clocked flip-flops

Flip-flops made with two gates, as in Figures 10.49 and 10.53, are known generically as *SR* (set–reset), or *jam-loaded*, flip-flops. You can force them into one state or the other whenever you want by just generating the right input signal. They're handy for switch debouncing and many other applications. But the most widely used form of flip-flop looks a little different. Instead of a pair of jam inputs, it has one or two “data” inputs and a single “clock” input. The outputs will change state or stay the same, depending on the levels at the data inputs when the clock pulse arrives.

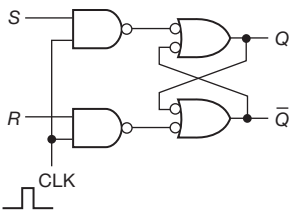


Figure 10.54. Clocked flip-flop: first approximation.

The simplest clocked flip-flop looks as shown in Figure 10.54. It's just our original flip-flop, with a pair of added gates (controlled by the clock) to enable the SET and RESET inputs. It is easy to verify that the truth table is

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	indeterminate

where Q_{n+1} is the Q output after the clock pulse and Q_n is

the output before the clock pulse. The basic difference between this and the previous flip-flops is that R and S should now be thought of as *data* inputs (as opposed to flip-flop set–reset inputs). What is present on R and S when a short clock pulse comes along determines what happens to Q . The R and S inputs are otherwise ignored.

This flip-flop has one awkward property, however. The output can change in response to the inputs during the time the clock is HIGH. In that sense it is still like the jam-loaded SR flip-flop (it's also known as a *transparent latch*: the output “sees through” to the input when the clock is HIGH). The full utility of clocked flip-flops comes with the introduction of slightly different configurations: the master–slave flip-flop, and the edge-triggered flip-flop.

A. Edge-triggered: the type-D flip-flop

These are by far the most popular kind of flip-flop. The logic level present at the D (data) input just before⁴⁶ a clock transition, or “edge,” determines the output state after the clock has changed.

Here's the truth table for the type- D flip-flop:

D	Q_{n+1}
0	0
1	1

You can think of the D -FF as copying the state of its input to its (latched) output. One of its many uses is, in fact, simply grabbing and holding a transient logic level,⁴⁷ as commanded by a separate clocking transition. These flip-flops are available as inexpensive packaged ICs and are always used in that form.

It is worth pausing for a moment to look at the innards of a D flip-flop, in order to understand what is going on. Figure 10.55 shows two circuit configurations, known as “master–slave” and “edge-triggered,” respectively. The master–slave configuration is probably easier to understand. Here's how it works.

While the clock is HIGH, gates 1 and 2 are enabled, forcing the master flip-flop (gates 3 and 4) to the same state as the D input: $M=D$, $M'=D'$. Gates 5 and 6 are disabled, so the slave flip-flop (gates 7 and 8) retains its previous state. When the clock goes LOW, the inputs to the master are disconnected from the D input, while the inputs of the slave are simultaneously coupled to the outputs of the master. The master thus transfers its state to the slave. No further

changes can occur at the output, because the master is now stuck. At the next rising edge of the clock, the slave will be decoupled from the master and will retain its state, while the master will once again follow the input.

The edge-triggered circuit behaves the same externally, but the inner workings are different. It is not difficult to figure them out. The particular circuit shown happens to be the ever-popular '74 rising-edge-triggered type- D flip-flop. Flip-flops are available with either rising- or falling-edge triggering.⁴⁸ (The preceding master–slave circuit transfers data to the output on the *falling* edge.) In addition, most flip-flops also have SET and CLEAR jam-type inputs. They may be set and cleared on HIGH or on LOW, depending on the type of flip-flop. Figure 10.56 shows four flip-flop examples. The wedge means “edge-triggered,” and the little circles mean “negation,” or complement. Thus the '74 is a dual type D rising-edge-triggered flip-flop with active-LOW jam-type SET and CLEAR inputs; the '4013 is a dual type- D rising-edge-triggered flip-flop with active-HIGH jam-type SET and CLEAR inputs; the '1G79 is a single type- D rising-edge-triggered flip-flop with no SET or CLEAR inputs and without a Q' output; and the '112 is a dual JK master–slave flip-flop with data transfer on the falling edge and with active-LOW jam-type SET and CLEAR inputs.

JK and type-T flip-flops

The JK and T flip-flops work on principles similar to those of the type- D flip-flop. Here are the truth tables:

J	K	Q_{n+1}	T	Q_{n+1}
0	0	Q_n	0	Q_n
0	1	0	1	Q'_n
1	0	1		
1	1	Q'_n		

Thus, if J and K are complements, Q will go to the value of the J input at the next clock edge. If J and K are both LOW, the output won't change. If J and K are both HIGH, the output will “toggle” (reverse its state after each clock pulse).

The T (toggling-type) flip-flop toggles at each clock if T is HIGH; T LOW makes it stick.

B. Divide-by-2

In digital circuits you often want to create a signal that toggles at a subdivision of some higher-frequency “clock” signal that is already present. For example, digital wrist-watches use as their timebase a 32,768 Hz crystal oscillator; that peculiar frequency is chosen because its 2^{15}

⁴⁶ This vague wording can, and will, be made precise, when we visit *setup* and *hold* times in §10.4.2C.

⁴⁷ Or perhaps a bunch of separate bits (e.g., an 8-bit data byte) in an array of D -flops (which is called a *register*).

⁴⁸ Sometimes loosely called *positive edge* and *negative edge*, respectively.

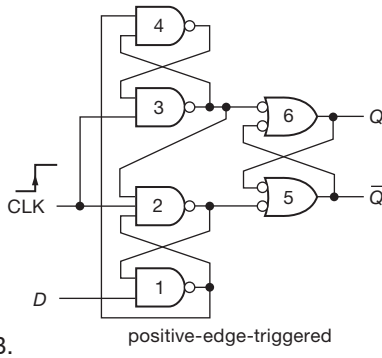
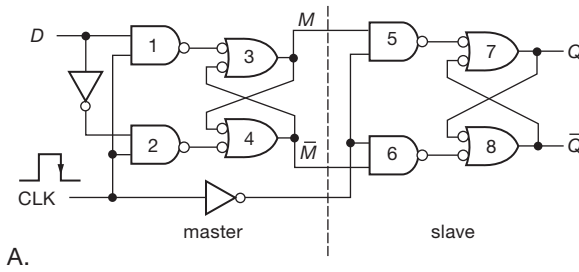


Figure 10.55. True clocked flip-flops: master–slave and edge-triggered type-*D* flip-flops.

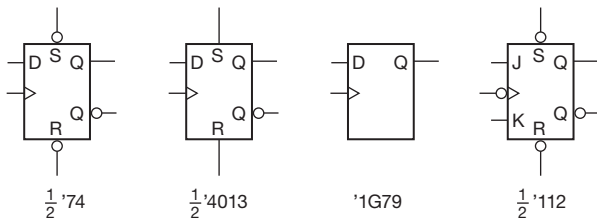


Figure 10.56. *D*-type and *JK* flip-flops.

subdivision is 1 Hz, just right for stepping the clock's second hand (or incrementing its displayed digital time). The basic trick here is to use the toggling capability of flip-flops: in Figure 10.57 the *D* flip-flop always sees, at its *D* input, the complement of its existing *Q* output. It therefore toggles at each clock pulse, generating an output at half the input clock frequency.

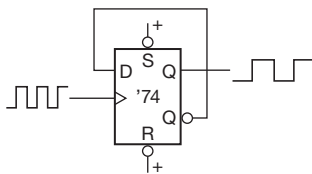


Figure 10.57. Toggling *D* flip-flop.

In Figure 10.58 you can see the essence of Verilog coding for the toggling *D* flip-flop.

```

wire CLKIN;
reg Q;
always @(posedge CLKIN)
    Q = ~Q;

```

Figure 10.58. Verilog code for the toggling *D* flip-flop.

C. Data and clock timing

This last circuit raises an interesting question: will the circuit fail to toggle, because the *D* input changes almost immediately after the clock pulse? In other words, will the circuit get confused, with such crazy things happening at its input? You could, instead, ask this question: exactly *when* does the *D* flip-flop (or any other flip-flop) look at its input, relative to the clock pulse? The answer is that there is a specified *setup time* t_s and *hold time* t_h for any clocked device. Input data must be present and stable from at least t_s before the clock transition until at least t_h after it, to guarantee proper operation. For the 74HC74, for instance, $t_s=20$ ns and $t_h=3$ ns (Figure 10.59). So, for the preceding toggling connection, the setup-time requirement is met if the output has been stable for at least 20 ns before the next clock rising edge. It may look as if the hold-time requirement is violated, but that's OK, also. The minimum *propagation time* from clock to output is 10 ns, so a *D* flip-flop connected to toggle as described is guaranteed to have its *D* input stable for at least 10 ns after the clock transition. Most devices nowadays have a zero hold-time requirement.

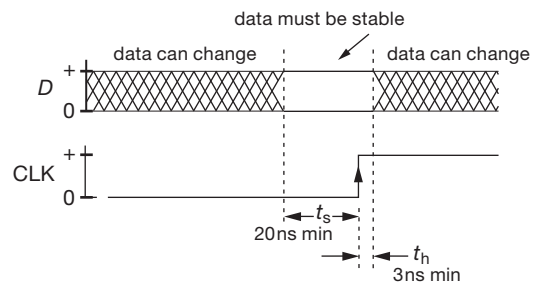


Figure 10.59. Data setup and hold times for the 74HC74 flip-flop.

D. Metastability

An interesting thing can happen if the level at the *D* input changes during the setup-time interval, namely a so-called *metastable* state in which the flip-flop can't make up its mind which state to go into. Figure 10.60 shows what happened when we deliberately violated the setup time for

a 74HC74 type-*D* flip-flop (operating from a +3.3 V supply), bringing D_{in} HIGH at the last moment: the Q output took its good old time deciding which state to go to.⁴⁹ The 'scope trace accumulated about 2 seconds of data, during which there were some events in which the delay from clock to Q output stretched out to nearly 50 ns, compared with its normal value of about 16 ns; when we ran it for several minutes, its best cliffhanger clocked in at 75 ns (nearly at the right-hand edge of the screen). Faster logic families show a correspondingly shorter delay, and it is claimed that some are designed to be “metastable resistant.”⁵⁰ We teased a 74LVC74 (at +3.3 V supply) into metastability (which occurred at $t_s \sim 0.4$ ns!), and measured a leisurely (!) propagation delay of 2–4 ns, compared with its frenetic normal delay of just 1.4 ns.⁵¹

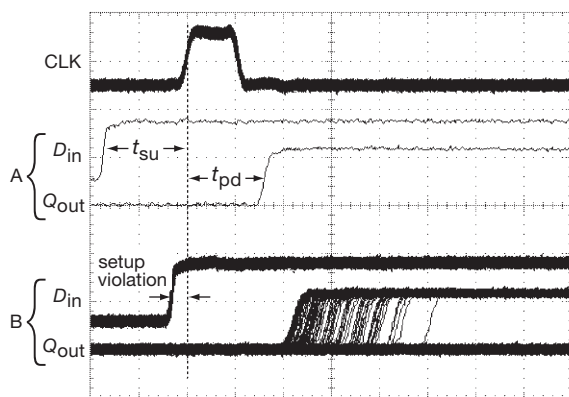


Figure 10.60. Timing violation produces metastability in clocked logic. Here a setup time violation at the D input of a 74HC74 (trace pair B) produces a much delayed output (Q_{out}) from CLK to Q compared with the normal delay of ~ 16 ns (trace pair A). Horizontal scale: 10 ns/div.

Just for fun we measured the lengthening of “decision time” for the 74HC74 flip-flop, as we tortured it with less and less setup time; Figure 10.61 shows the results.

E. Divide by more

By cascading several toggling flip-flops (connect each Q output to the next clock input), it is easy to make a divide-by- 2^n , or binary, counter. Figure 10.62 shows a three-stage

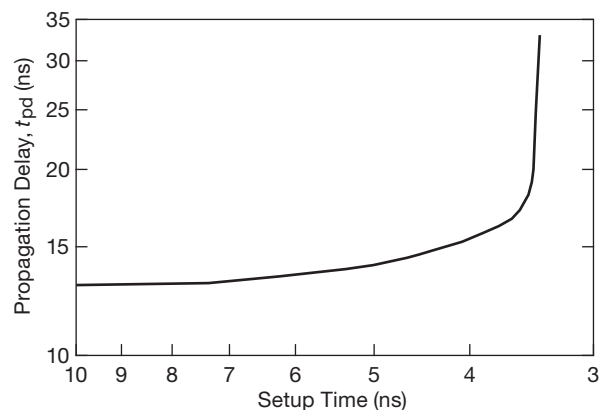


Figure 10.61. Twisting the dragon’s tail: metastable propagation delay versus (violated) setup time for the 74HC74.

ripple counter, which is a divide-by-8: the output waveform from the last flip-flop is a square wave whose frequency is $1/8$ of the circuit’s input clock frequency. Such a circuit is called a *counter* because the data present at the three Q outputs, considered as a single 3-bit binary number, goes through a binary sequence from 0 to 7, incrementing after each input pulse. (We’ve used the Q' outputs to clock successive stages, to make it count *up* rather than *down*.⁵²)

Figure 10.63 shows measured waveforms for this counter, zipping along at 50 MHz (the curved arrows were added to indicate what causes what, to aid in understanding): you can see the binary sequence of 3-bit numbers (0–7), where Q_A is the LSB, and Q_C is the MSB. You can also see that there is successive delay going from stage to stage (hence “ripple”).⁵³

In practice, the simple scheme of cascading counters by connecting each Q output to the next clock input creates certain problems related to the cascaded delays as the signal ripples down through the chain of flip-flops, and a *synchronous* scheme (in which all clock inputs see the same clocking signal) is usually better.⁵⁴ We’ll see soon how to make a 3-bit synchronous counter.

The counter is a useful function, with many versions

⁴⁹ The abrupt Q output, buffered by the output stage, conceals the more interesting internal metastable behavior, in which the (unbuffered) voltage hovers between LOW and HIGH, balanced on a knife edge, trying to decide which way to fall.

⁵⁰ Metastable-*resistant* is not the same thing as metastable-*proof*; it’s like water-resistant versus waterproof.

⁵¹ The LVC family is a good overall choice for operation from 1.8–3.3 V, though by no means the fastest kid on the block (see Figure 10.26).

⁵² There are two other configurations that produce an *up-counter*: (1) use the Q outputs to clock successive stages, but use the Q' signals as your binary outputs; or (2) use the Q outputs to clock successive falling-edge-triggered stages.

⁵³ In fact, the ripple delay is large enough here so that the count at any instant (i.e., a vertical slice through the traces) is never correct. This does not matter, however, if the counter is used only to generate an output of subdivided frequency, or if the counter is stopped prior to readout.

⁵⁴ However, if all you need is a 2^n subdivision of an input clock, with no

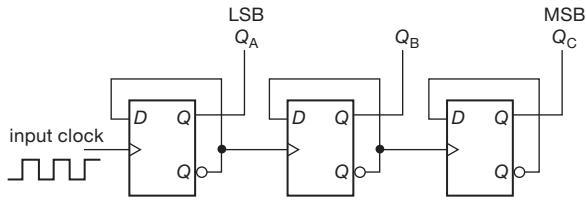


Figure 10.62. Three-stage binary ripple counter.

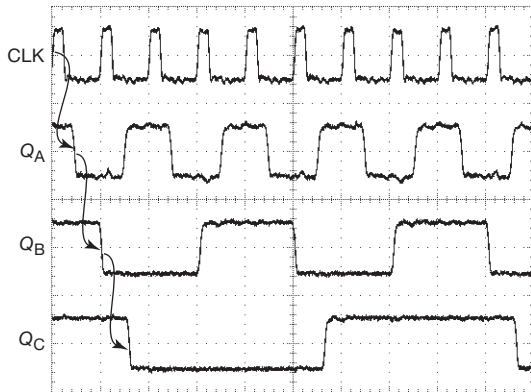


Figure 10.63. Ripple-counter waveforms. This 'scope trace (at 4 V/div vertical, 20 ns/div horizontal) shows a cascade of rising-edge triggered 74HC74 *D* flip-flops, clocked at 50 MHz, displaying successive ~10 ns delays per stage. The counter, beginning at maximum count (1, 1, 1), is clocked to all zeros by the rising edge of CLK at the extreme left edge. See Figure 10.71 for analogous waveforms of a fully-synchronous counter.

available as standard logic, including 4-bit, BCD, and multidigit counting formats. By cascading several such counters and displaying the count on a numeric display device (e.g., an LED digital display), you can easily construct an event counter. If the input pulse train to such a counter is gated for exactly 1 second, you've got a frequency counter that displays frequency (cycles per second) by simply counting the number of cycles in a second.

10.4.3 Combining memory and gates: sequential logic

Having explored the properties of flip-flops, let's see what can be done when they are combined with the combinational (gate) logic we discussed earlier. Circuits made with gates and flip-flops constitute the most general form of digital logic.

particular phase relation to the input, a ripple counter is perfectly good; it is also simpler, and will operate to a higher maximum frequency.

A. Synchronously clocked systems

As we hinted in the previous subsection, sequential logic circuits in which there is a common source of clock pulses driving all flip-flops have some very desirable properties. In such a *synchronous system*, all action takes place just after each clocking pulse, based on the stable levels present just before the next clock pulse. The system reaches its next stable state before the next clock pulse; it's a nice way of managing feedback; and, by clocking all flip-flops simultaneously, it takes advantage of the relative lack of digital noise just before each clock pulse (the calm before the storm).

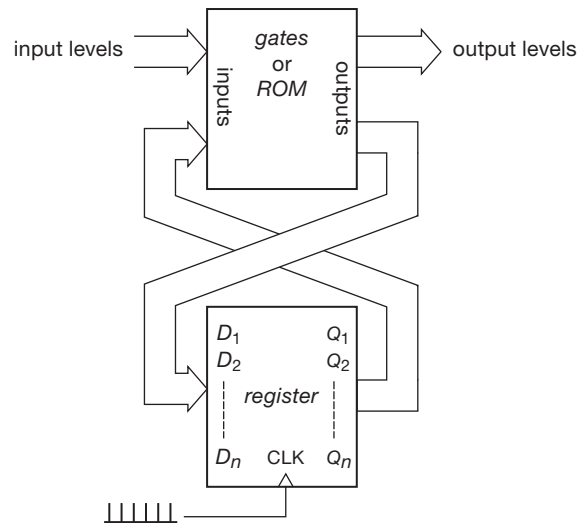


Figure 10.64. The classic sequential "state machine": a set of clocked flip-flops (a *register*) plus combinational logic. This scheme is easily implemented with single-chip programmable logic devices (PLDs or FPGAs).

Figure 10.64 shows the general scheme. The flip-flops have all been combined into a single *register*, which is nothing more than a set of type-*D* flip-flops with their clock inputs all tied together, and their individual *D* inputs and *Q* outputs brought out; i.e., each clock pulse causes the levels present at the *D* inputs to be transferred to the respective *Q* outputs. The box full of gates looks at both the *Q* outputs and whatever input levels are applied to the circuit and generates a new set of *D* inputs and logic outputs. This simple-looking scheme is extremely powerful; it is the basis for general digital processors. Let's look at an example.

B. Example: divide-by-3

Let's design a synchronous divide-by-3 circuit with two *D* flip-flops, both clocked from the input signal. In this case,

D_1 and D_2 are the register inputs, Q_1 and Q_2 are the outputs, and the common clock line is the master clocking input (Figure 10.65). The trick is to rig up gating so that the D inputs are presented with the desired next state.

1. Choose the three states. Let's use

Q_1	Q_2
0	0
0	1
1	0
0	0

(i.e., first state)

2. Find the combinational logic network outputs necessary to generate this sequence of states; i.e., figure out what the D inputs have to be to get those outputs:

Q_1	Q_2	D_1	D_2
0	0	0	1
0	1	1	0
1	0	0	0

3. Concoct suitable combinational logic (gates), using available outputs, to produce those D inputs. In general, you can use a "lookup table" (LUT) in ROM (read-only memory), programmed to hold the next-state D values (and addressed by the current Q value, along with any external inputs).⁵⁵ This example is simple, so you can do it with one logic gate: you can see by inspection that

$$\begin{aligned} D_1 &= Q_2, \\ D_2 &= (Q_1 + Q_2)', \end{aligned}$$

from which the circuit of Figure 10.66 follows.

It is easy to verify that the circuit works as planned. Because it is a synchronous counter, all outputs change simultaneously (unlike the ripple counter). In general, synchronous (or "clocked") systems are desirable, since susceptibility to noise is improved: things have settled down by the time of the next clock pulse, so circuits that look at their inputs only at clock edges aren't troubled by capacitively coupled interference from other flip-flops, etc. A further advantage of clocked systems is that transient states (caused by delays, so that all outputs don't change simultaneously) don't produce false output, because the system is insensitive to what happens just *after* a clock pulse. We'll see some examples later.

⁵⁵ This example has no logic inputs, as you would have with, for example, an UP/DOWN counter, or a counter with a RESET input.

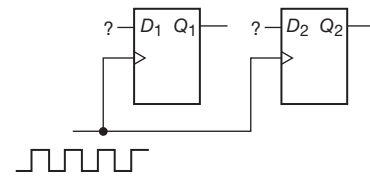


Figure 10.65. Divide-by-3: need logic!

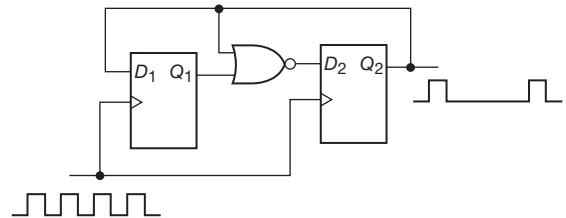


Figure 10.66. Synchronous divide-by-3.

C. Excluded states

What happens to the divide-by-3 circuit if the flip-flops somehow get into the state $(Q_1, Q_2) = (1, 1)$? This can easily happen when the circuit is first turned on, since the initial state of a flip-flop is anyone's guess. From the diagram, it is clear that the first clock pulse will cause it to go to the state $(1, 0)$, from which it will function as before. It is important to check the excluded states of a circuit like this, since it is possible to be unlucky and have it get stuck in one of those states. (Alternatively, and better, the initial design procedure can include a specification of all possible states.) A useful diagnostic tool is the *state diagram*, which for this example looks like Figure 10.67. Usually you write the conditions for each transition next to the arrows, if other variables of the system are involved. Arrows may go in both directions between states, or from one state to several others.

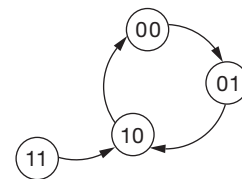


Figure 10.67. State diagram: divide-by-3.

The Verilog code for this circuit (this time a complete "module") is in Figure 10.68.

Exercise 10.23. Design a synchronous divide-by-3 circuit using two JK flip-flops. It can be done (in 16 different ways!) without any gates or inverters. *One hint:* when you construct the table of required J_1, K_1 and J_2, K_2 inputs, keep in mind that there are

```

module divideBy3(CLKIN,Q1,Q2);
    input CLKIN;
    output Q1, Q2;
    reg Q1, Q2;
    always @(posedge CLKIN)
        begin
            Q1 <= Q2;           // the <= symbol is called a "nonblocking assignment"
            Q2 <= ~(Q1|Q2);     // it makes all steps happen at once, not sequentially
        end
endmodule

```

Figure 10.68. Verilog code for divide-by-3 circuit.

two possibilities for J, K at each point. For instance, if a flip-flop output is to go from 0 to 1, $J, K = 1, X$ ($X = \text{don't care}$). Finally, check to see if the circuit will get stuck in the excluded state (of the 16 distinct solutions to this problem, 4 will get stuck and 12 won't).

Exercise 10.24. Design a synchronous 2-bit UP/DOWN counter: it has a clock input, and a control input (U/D'); the outputs are the two flip-flop outputs Q_1 and Q_2 . If U/D' is HIGH, it goes through a normal binary counting sequence; if LOW, it counts backward – $Q_2Q_1 = 00, 11, 10, 01, 00, \dots$

D. State diagrams as design tools

The state diagram can be quite useful when designing sequential logic, particularly if the states are connected together by several paths. In this design approach, you begin by selecting a set of unique states of the system, giving each a name (i.e., a binary address). You will need a minimum of n flip-flops, or bits, where n is the smallest integer for which 2^n is equal to or greater than the number of distinct states in the system. Then you set down all the rules for moving between states, i.e., all possible conditions for entering and leaving each state. From there it is a straightforward (but perhaps tedious) job to generate the necessary combinational logic, because you have all possible sets of Q 's and the set of D 's that each leads to. Thus you have converted a sequential design problem into a combinational design problem.⁵⁶ Figure 10.69 shows a real-world example. Note that there may be states that don't lead to others, e.g., "with diploma."

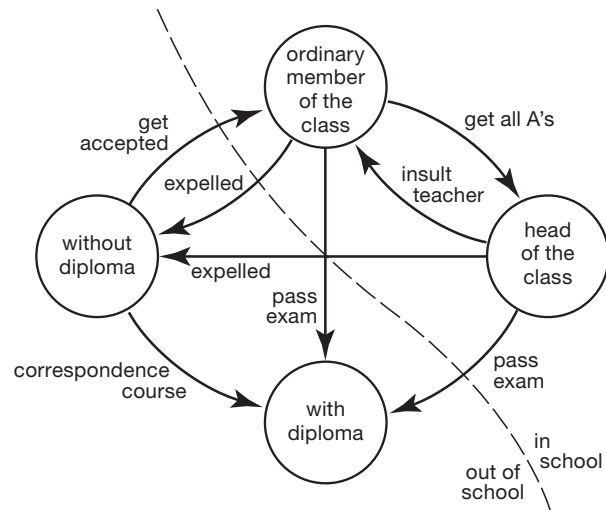


Figure 10.69. State diagram: going to school.

E. State machine design

Hardware description languages, used as algebraic entry tools for both programmable logic (cPLDs and FPGAs) and for full-custom integrated circuits (ASICs), provide a straightforward route to specify state machines. They include *if/elseif/else* statements that let you specify conditions for going to the next state, and for the outputs, given the current state and the inputs. The HDL software translates these specifications into a logic circuit, implemented with the usual gates and flip-flops.

One further point about state machines: the block diagram of Figure 10.64 allows for two possibilities, namely that (a) the outputs may depend on only the current state (defined by the Q 's), and in fact may consist simply of the Q 's themselves; or (b) for any given state of the Q 's, the output may depend also upon the inputs, which are available to the combinational logic in the block labeled "gates or ROM." These are known as *Moore* and *Mealy* state

⁵⁶ You can always take the shortcut of using a lookup table, in the form of a ROM. Alternatively, there are design tools (e.g., hardware description languages (HDLs) used for programmable logic) that will minimize and instantiate the needed logic. Best of all, these HDLs always include an input syntax that lets you simply specify the states and their transition rules, so that you don't even have to figure out the necessary logic.

machines, respectively.⁵⁷ The Moore machine changes state only on clock edges, and, if the Q 's themselves are considered the outputs, they are strictly synchronous. The Mealy machine responds asynchronously to input changes, regardless of the clock; it generally requires fewer flip-flops, because there are multiple possible outputs corresponding to a single state (defined by the Q 's).

F. Synchronous binary counter

We promised we'd revisit our original 3-bit ripple counter (§10.4.2E) to make it fully synchronous. Figure 10.70 shows how it's done: the LSB always toggles, easily done by feeding back to its D an inverted Q . For higher-order bits the rule for counting in binary is to toggle only when all lower-order bits are 1s. Because an XOR is an "optional inverter," this is easily done by feeding each D input from an XOR, one of whose inputs comes from the corresponding Q , and the other from an AND of all lower-order Q 's. It's worth seeing it also in a hardware description language:⁵⁸

```
QA.d = !QA.q
QB.d = QB.q $ QA.q
QC.d = QC.q $ (QB.q & QA.q)
```

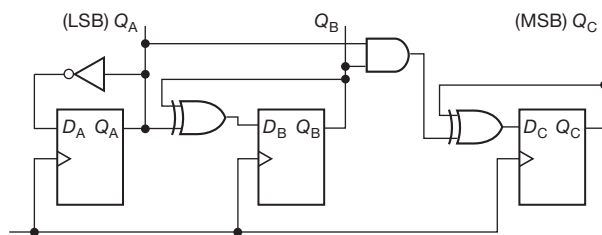


Figure 10.70. 3-bit synchronous binary counter.

As promised, the Q 's all clock simultaneously,⁵⁹ as you can see in Figure 10.71, which shows measured waveforms for a 3-bit synchronous binary counter clocked at the same rate as the ripple counter of Figure 10.63. Here it is easy to see that the Q 's of the successive clocked states are simply the numbers 0 to 7 in 3-bit binary.



Figure 10.71. In contrast to a ripple counter (Figure 10.62), a *synchronous* counter (or, indeed, any synchronous system) has all flip-flops clocked from a common source. Here a 74HC161 4-bit synchronous counter (rising-edge triggered) displays a CLK-to- Q delay of about 14 ns, constant for all Q 's, thus producing coincident changes. (Same conditions as Figure 10.63)

G. PLDs for state machine design

Programmable logic devices have just what you need for state machines – plenty of flip-flops and plenty of logic, all on one configurable (hence “programmable”) chip. And the programming software includes tools that make state-machine design straightforward. We'll have much more to say about these remarkable devices in §10.5.4, and in the next chapter.

10.4.4 Synchronizer

An interesting application of flip-flops in sequential circuits is their use in a *synchronizer*. Suppose you have some external control signal coming into a synchronous system that has clocks, flip-flops, etc., and you want to use the state of that input signal to control some action. For example, a signal from an instrument or experiment might signify that data are ready to be sent to a computer. Because the experiment and the computer march to the beats of different drummers (in fancy language you would say they are *asynchronous* processes), you need a method to restore order between the two systems.

A. Example: pulse synchronizer

As an example, let's reconsider the circuit in which a debouncer flip-flop gated a pulse train (Figure 10.51 in §10.4.1A). That circuit enables the gate whenever the switch is closed, regardless of the phase of the pulse train being gated, so that the first or last pulse may be shortened. The problem is that the switch closure is asynchronous

⁵⁷ It's easy to get them mixed up; perhaps the mnemonic “Moore↔OutputsOnly” is helpful.

⁵⁸ We've chosen the primitive ABEL, rather than something sophisticated like Verilog, because it is “closer to the metal”: it lets you specify D inputs and Q outputs, etc. In ABEL the XOR is written as \$.

⁵⁹ Well, almost. At a fussier level of scrutiny you can worry about timing *skew*, the (relatively small) scatter of propagation times, among the various flip-flops, from the common clocking edge to the various Q outputs.

with the pulse train. In some applications it is important to have only *complete* clock cycles, and that requires a synchronizer circuit like that in Figure 10.72. When the switch creates a debounced HIGH on the D input, Q stays LOW until the next falling edge of the input pulse train. In that way, only complete pulses are passed by the AND gate. In the waveforms the curved arrows are drawn to show what causes what. You can see, for instance, that the transitions of Q occur slightly *after* the falling edges of the input.

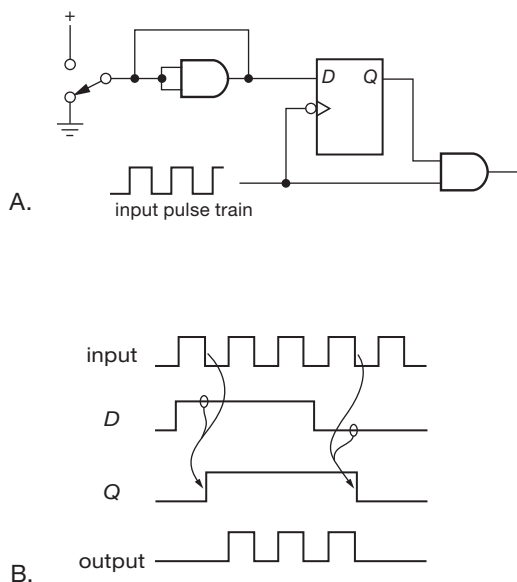


Figure 10.72. Pulse train synchronizer: A. circuit; B. timing diagram.

B. Logic races and glitches

This example brings up a subtle but extremely important point: what would happen if a *rising-edge-triggered* flip-flop were used instead? If you analyze it carefully, you'll find that things work OK at the beginning of the pulse train, but that a bad thing happens at the end (Figure 10.73). A short spike (or "glitch" or "runt pulse") gets through because the final AND gate isn't disabled until the flip-flop output has a chance to go LOW, a delay of about 20 ns for HC family logic. This is a classic example of a *logic race*. With some care these situations can be avoided, as the example shows.⁶⁰ Glitches are terrible things to have running through your circuits. Among other things, they're hard to

see on an oscilloscope, and you may not know they are present. They can clock subsequent flip-flops erratically, and they may be widened – or narrowed to extinction – by passage through gates and inverters.

Exercise 10.25. Demonstrate that the preceding pulse-synchronizer circuit (Figure 10.72) does not generate glitches.

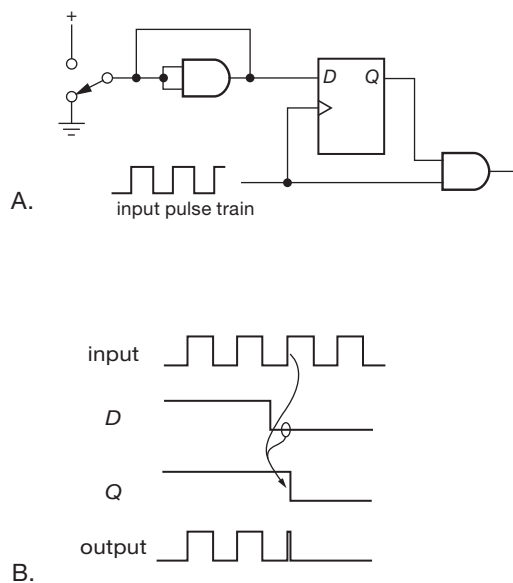


Figure 10.73. A logic race can generate a "runt pulse."

A few comments about synchronizers: the input to the D flip-flop can come from other logic circuitry, rather than from a debounced switch. There are applications in computer interfacing, etc., where an asynchronous signal must communicate with a clocked device; in such cases clocked flip-flops or synchronizers are ideal. In this circuit, as in all logic, unused inputs must be handled properly. For instance, SET and CLEAR must be connected so that they are not asserted (for a '74, tie them HIGH; for a 4013, they are grounded). Unused inputs that have no influence on the outputs can be tied to either level.

clocks the flip-flop. OK, you say, "I'll clock the rising-edge flip-flop with the input pulse train, and I'll pass that same input through an inverter on the way to the AND gate." Sounds fine, but there's a new problem, namely a logic hazard at the *beginning* of the synchronized output train. Challenge yourself by figuring it out, and drawing a circuit that works properly with a rising-edge flip-flop.

⁶⁰ That example required a falling-edge-triggered flip-flop, which is a rare species – but a rising-edge flip-flop creates the glitch! What to do? With some thought you will realize that the glitch-free condition is this: *the clock level that enables the gate must precede the edge that*

10.4.5 Monostable multivibrator

In §7.2.2 we introduced what is really a *mixed-signal* device (combined analog and digital), the monostable multivibrator (also known as a “one-shot,” accent on the *one*); see Tables 7.3 and 7.4 on pages 462 and 463. Although we counseled some caution when considering monostables, there are times when they do just what you want. Let’s interrupt our narrative here to look at a nice example, namely, a triggerable pulse generator.

A. Pulse-generator example

Figure 10.74 shows a square-wave generator with independently settable rate and duty cycle (ratio of HIGH to LOW), along with an input that permits an external signal to start and stop the pulse train synchronously. Current source U_1 – Q_1 generates a ramp at C_1 , with slew rate proportional the resistance of panel potentiometer R_2 . When C_1 ’s voltage reaches the upper comparator’s threshold of 3.0 V, the one-shot is triggered and generates a 100 ns LOW pulse, putting n -channel MOSFET Q_2 into conduction and discharging the capacitor. C_1 therefore has a sawtooth waveform going from ground to +3 volts, with the rate set by potentiometer R_2 . The lower comparator generates an output square wave from the sawtooth, with duty cycle adjustable linearly between 2% and 98% via R_7 . Both comparators have a few millivolts of hysteresis (R_{10} and R_{11}) to prevent noise-induced multiple transitions. The TLV3502 is a fast (4.5 ns) CMOS dual comparator with input common-mode range to both rails (ground and +5 V), and rail-to-rail outputs (see Table 12.2 on page 813 for more comparator types).

A feature of this circuit is its ability to synchronize (start/stop) to an externally applied control level. The ENABLE input lets the driven circuit start the oscillator at a predictable phase (the time of an output pulse’s falling edge), and command the oscillator to stop after the next complete pulse.

A few instructive details (which can be skipped over on a first reading).

- The additional input to the 3-input NAND from the comparator output ensures that the circuit won’t get stuck with C_1 charged up.
- The one-shot pulse width has been chosen long enough to ensure that C_1 is fully discharged during the pulse: you can estimate the discharge time from looking at both the 2N7000’s saturation drain current with 5 V gate drive (about 350 mA), and its R_{on} (5 Ω) when the discharge is nearly complete; they correspond to approximate discharge times of 50 ns and 17 ns, respectively, from which we conservatively set a fixed discharge time of 100 ns.

- The 50 Ω resistor at the output provides “source termination” for a 50 Ω cable (see Appendix H).
- We’ve arranged things so that the top of the sawtooth waveform corresponds to the short duty-cycle setting; that’s because the waveform has a 100 ns flat bottom (during discharge), whereas the top is an accurate sawtooth.
- Note that the oscillation frequency is, to a good approximation, independent of supply-voltage variations, being set ratiometrically (both the charging current and the peak amplitude are set as a fraction of V_+ , nominally 5 V).
- The frequency-setting voltage applied to U_1 is bypassed to the *positive* rail, because that is the reference for the current source; at the low-frequency end of the range, the voltage at R_{15} is just 5 mV below the positive rail, thus rather sensitive to supply-voltage noise.
- We’ve specified a log-taper potentiometer for the frequency control R_2 ; otherwise the low-frequency end of the range gets squished.

10.4.6 Single-pulse generation with flip-flops and counters

In §7.2.2 we discussed the generation of pulses and time delays with monostable multivibrators, and some reasons to use caution when considering whether to use these partly-analog devices. When a clocking signal is present (which is almost always), there are purely digital alternatives. Figure 10.75 shows how to make a single glitch-free pulse whose width equals one clock cycle. You’ll have fun drawing the timing diagram for this circuit – take the challenge!

Exercise 10.26. Take the challenge!

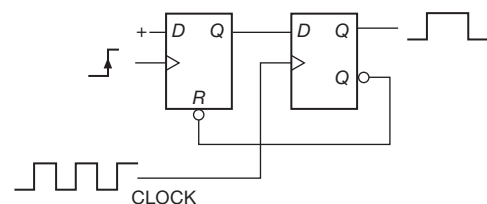


Figure 10.75. Single-pulse generator.

Figure 10.76 shows another case in which flip-flops and counters (cascaded toggling flip-flops) can be used in place of a monostable to generate a long output pulse. The ’4060 is a 14-stage CMOS binary ripple counter (14 cascaded flip-flops). A rising edge at the input brings Q HIGH, enabling the counter. After 2^{n-1} clock pulses, Q_n goes HIGH,

is similar, but has a reset instead of three-state outputs. For larger data widths you can get the '16374 (16-bit) and '32374 (32-bit) *D* registers. Figure 10.77 shows a quad *D* register with both true and complemented outputs. The Verilog code that describes it is in Figure 10.78.

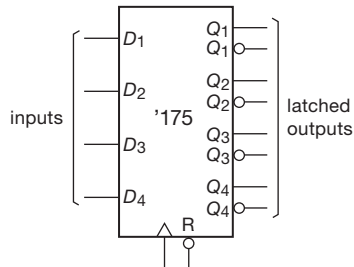


Figure 10.77. '175 4-bit *D* register.

```
// 4-bit D-register with true and complemented
// outputs, and with active-LOW asynchronous reset
reg [3:0] D;
reg [3:0] Q;
reg [3:0] QBAR;
wire CLKIN;
wire RESETBAR;
always @(posedge CLKIN) or negedge RESETBAR
    if (!RESETBAR)
        begin
            Q <= 4'b0000;
            QBAR <= 4'b1111;
        end
    else
        begin
            Q <= D;
            QBAR <= ~D;
        end
end
```

Figure 10.78. Verilog code for a quad *D*-register with true and complemented outputs.

The term “latch” is usually reserved for a special kind of register: one in which the outputs follow the inputs when enabled, and hold the last value when disabled. But the term “latch” has become ambiguous with use, so the terms “transparent latch” and “type-*D* register” are often used to distinguish these closely related devices. As an example, the '573 is the octal transparent latch analogous to the '574 *D* register, complete with three-state outputs. The 16-bit and 32-bit versions are called '16373 and '32373.

Some variations on the latch or register are as follows: (a) random-access memories (RAMs), which let you write to, and read from, a (usually large) set of registers, but only

one (or at most a few) at a time; RAM ICs come in sizes up to 1 Gbyte or more, and are used primarily for memory in microprocessor systems (see Chapter 14); (b) addressable latches, a multibit latch that lets you update individual bits while keeping the others unchanged; (c) a latch or register built into a larger chip, for example a digital-to-analog converter; such a device needs the input applied only momentarily (with appropriate clocking edge), since an internal register can hold the data.

When choosing a register or latch, look for important features such as input enable, reset, three-state outputs, and *broadside* pinout (inputs on one side of the chip, outputs on the other); the latter is convenient when you are laying out a printed-circuit board (PCB).

10.5.2 Counters

As we mentioned earlier, it is possible to make a *counter* by connecting flip-flops together. There is available an amazing variety of such devices as single chips, some of which are listed in Table 10.5. Here are some of the features to look for.

A. Size

You can get BCD (divide-by-10) and binary (or *hexadecimal*, divide-by-16) counters in the popular 4-bit category. There are larger counters, up to 24 bits (not all of them available as outputs); the 74LV8154 is a nice example – it has a pair of 16-bit synchronous counters with output registers and separate clocking inputs, and an 8-bit three-state output that forwards any selected byte. There are also modulo-*n* counters that divide by an integer *n*, with the modulus *n* specified as an input. For some applications (e.g., timing) you don’t care about the intermediate bits, you just want a lot of internal stages, provided by chips like the ICM7240–60, MC14541, and MC14536; see §7.2.4D. You can always cascade counters (including synchronous types) to get more stages.

B. Clocking

An important distinction is whether the counter is a ripple counter or a synchronous counter. The latter clocks all flip-flops simultaneously, whereas in a ripple counter each stage is clocked by the output of the previous stage (Figures 10.62 and 10.63). Ripple counters generate transient states because the earlier stages toggle slightly before the later ones. For instance, a ripple counter going from a count of 7 (0111) to 8 (1000) goes through the states 6, 4, and 0 along the way. This doesn’t cause trouble in well-designed circuits, but it would in a circuit that used gates to look for

Table 10.5 Selected Counter ICs^a

Part number (74xxx)	Supply voltage		Bits	Synchronous?	U/D	BCD	Sync clear	$f_{\max}$		DIP avail?
	min (V)	max (V)						min (MHz)	@ V_{CC} (V)	
HC4024	2.0	6.0	7	no	-	-	-	30	4.5	●
HC4040	2.0	6.0	12	no	-	-	-	30	5	●
VHC4040	2.0	6.0	12	no	-	-	-	150	5	-
HC4060	2.0	6.0	14 ^b	no	-	-	-	28	4.5	●
LV4060	1.2	5.5	14 ^b	no	-	-	-	99 ^C	3.3	(●)
HC40103	2.0	6.0	8	●	D	'102	●	15	4.5	●
74HC161	2.0	6.0	4	●	-	'162	'163	30	4.5	●
74AC161	1.5	6.0	4	●	-	-	'163	90 ^C	3.3	-
74LV161	2.0	5.5	4	●	-	-	'163	165 ^C	3.3	-
74LVC161	1.2	3.6	4	●	-	-	'163	200 ^C	3.3	-
74HC191	2.0	6.0	4	●	●	'192	'193	30	4.5	●
74AC191	1.5	6.0	4	●	●	-	-	133 ^C	5	-
74HC590	2.0	6.0	8 ^d	●	-	-	-	33	4.5	●

Notes: (a) all are binary, async reset, count up, unless otherwise marked.
(b) no output pins for bits 0, 1, 2, and 10. (c) typical. (d) with 3-state outputs.

a particular state (this is a good place to use something like a *D* register, so that the counter output state is examined only at a safe clock edge). Ripple counters are slower than synchronous counters, because of the accumulated propagation delays. That is, it takes longer for all bits to “settle” to their next state; on the other hand, a ripple counter has a higher maximum counting rate (for the same speed of flip-flop). Ripple counters clock on falling edges for easy expandability (by connecting the *Q* output of one counter directly to the clock input of the next); synchronous counters clock on the rising edge.

We favor the '160–'163 family of 4-bit synchronous counters for most applications that don't require a special feature (Figure 10.79). The four family members include BCD and binary, each available with either synchronous or asynchronous reset.⁶¹ They can be parallel loaded, and are easily cascaded via the carry output and enable inputs.

C. Up/down

Some counters can count in either direction, under control of some inputs. The two possibilities are (a) a *U/D'* input

⁶¹ With the reset input *R'* asserted, a counter with synchronous reset waits until the next rising clock to reset, whereas a counter with asynchronous reset clears upon assertion of the *R'* input, regardless of the state of the clock.

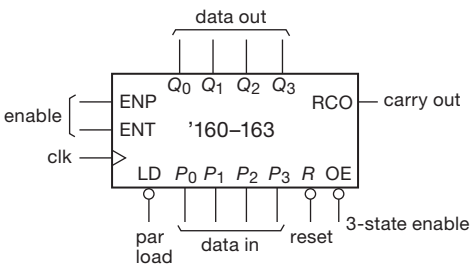


Figure 10.79. '160–'163 synchronous counters.

that sets the direction of count and (b) a pair of clocking inputs, one for UP, one for DOWN. Examples are the '191 and '193, respectively. The '579 and '779 are useful 8-bit up/down counters.

D. Load and clear

Most counters have data inputs so that they can be pre-set to a given count. This is handy if you want to make a modulo-*n* counter, for example. The load function can be either synchronous or asynchronous: the '160–'163 have synchronous load, which means that data on the input lines are transferred to the counter coincident with the next clock rising edge, if the *LOAD'* line is also asserted LOW; the '190–'193 are asynchronous, or *jam-load*, which means

that input data are transferred to the counter when LOAD' is asserted, independent of the clock. The term “parallel load” is sometimes used, since all bits are loaded at the same time.

The CLEAR (or RESET) function is a form of presetting. The majority of counters have a jam-type (asynchronous) CLEAR function, though some have synchronous CLEAR; for example, the '160/'161 are jam CLEAR, whereas the '162/'163 are synchronous CLEAR.

Example: divide-by-3 (again)

Let's pause for a moment to look at this business of synchronous versus asynchronous control signals. Neither is “better” (after all, the '160–'163 family gives you both choices, and at the same price) – the choice depends on the application. Let's imagine we want to conscript a '161/'163-style 4-bit synchronous binary counter into service as a divide-by-3 (a challenge we took on earlier, with discrete flip-flops, in §10.4.3B). Because these are *up*-counters only, we'll use a NAND gate to detect the state count = 3, and use its active-LOW output to assert the RESET' input of the counter. So it should count 0, 1, 2, and the next clock takes it to a count of 3, whereupon it is immediately reset. Because we want the reset to happen right away, we've chosen the '161, with its asynchronous RESET' input (Figure 10.80A).

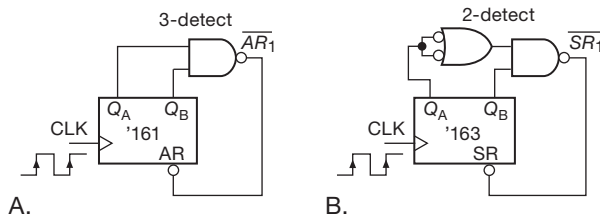


Figure 10.80. Two divide-by-3 circuits, from 4-bit synchronous counters: A. with asynchronous reset ('161); B. with synchronous reset ('163). We built both, with LV-A family logic running at 3.3 V; waveforms in Figures 10.81 and 10.82.

Sounds fine, and it does work, more or less. But there's a bit of a problem: look at Figure 10.81, which shows measured waveforms, as we clocked it at about 12 MHz. You can see it clock successively through states 0, 1, and 2; then, just past halfway across the trace you see it reach state 3 (Q_A and Q_B both HIGH), whereupon the NAND gate generates a LOW pulse (AR'_1), which resets the counter to 0. The problem is that the RESET' pulse contains the seeds of its own destruction, so to speak: it brings the count to zero, which makes it promptly disappear. So it's possible for the pulse to be shorter than the minimum reset pulse

width, which could result in incomplete resetting of all the counter's flip-flops.⁶²

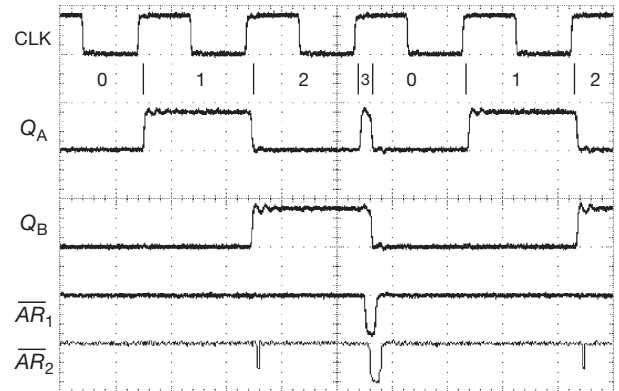


Figure 10.81. Measured waveforms from the divide-by-3 circuit of Figure 10.80A. The modified reset pulse AR'_2 resulted from 39 pF additional loading on Q_A . Horizontal: 40 ns/div. Vertical: 4 V/div.

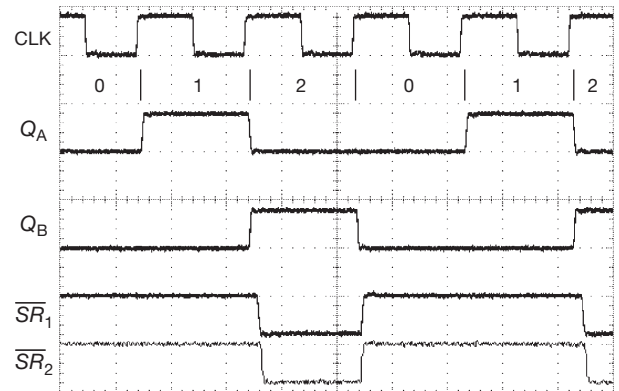


Figure 10.82. Measured waveforms from the divide-by-3 circuit of Figure 10.80B; same conditions as Figure 10.80A.

The scope trace shows another potential hazard of using asynchronous reset here: the trace labeled AR'_2 is the NAND gate output when the Q_A output sees more capacitive loading than Q_B (we put 39 pF to ground); that delayed Q_A enough to produce a transient false “3” state during the 1→2 transition. In this circuit it wasn't sufficient to prematurely reset the counter... but it sure looks bad!

The better solution is *synchronous* reset (Figure 10.80B), where the counter is reset at the rising clock *following* assertion of its RESET' input. That means we need to detect count = 2 (rather than 3; i.e., $n-1$ rather

⁶² Even if it works reliably in practice, you don't want little runt pulses running around your circuit, right?

than n).⁶³ Figure 10.82 shows the waveforms, free of glitches and runt pulses (even with extra capacitive loading of Q_A , shown as the reset waveform SR'_2).

We'll see this issue again in connection with the n -pulse generator (§10.6.3).

E. Other counter features

Some counters feature latches on the output lines; these are always of the transparent type, so the counter can be used as if no latch were present. (Keep in mind that any counter with parallel-load inputs can function as latch, but you can't count at the same time as data are held, as you can with a counter/latch chip.) The combination of counter plus latch is sometimes quite convenient, e.g., if you want to display or output the previous count while beginning a new counting cycle. In a frequency counter this would allow a stable display, with updating after each counting cycle, rather than a display that repeatedly gets reset to zero and then counts up.

There are counters with three-state outputs. These are great for applications where the digits (or 4-bit groups) are multiplexed onto a bus for display or transfer to some other device. Examples are the '560/1, '590, and '779; the latter is an 8-bit synchronous binary counter whose three-state outputs also serve as parallel inputs; by sharing input-output lines, the counter fits in a 16-pin package. The '593 is similar, but in a 20-pin package. If you want a counter to use with a display, there are some that combine counter, latch, 7-segment decoder, and driver on one chip; an example is the 74C926 4-digit counter.⁶⁴

10.5.3 Shift registers

If you connect a series of flip-flops so that each Q output drives the next D input, and all clock inputs are driven simultaneously, you get what's called a *shift register*. At each clock pulse the pattern of 0s and 1s in the register shifts to

the right, with the data at the first D input entering from the left. As with flip-flops, the data present at the serial input just prior to the clock pulse are entered, and there is the usual propagation delay to the outputs. Thus they may be cascaded without fear of a logic race. Shift registers are very useful for conversion of parallel data (n bits present simultaneously, on n separate lines) to serial data (one bit after another, on a single data line), and vice versa. They're also handy as memories, particularly if the data are always read and written in order. As with counters and latches, shift registers come in a pleasant variety of prefab styles. The important things to look for are the following.

A. Characteristics

Size and Organization

The 4-bit and 8-bit registers are standard, with some larger sizes available (up to 64 bits or more).⁶⁵

Shift registers are usually 1 bit wide, but there are also dual-, quad-, and hex-width registers. Most shift registers only shift right, but there are bidirectional registers like the '194 and '299.

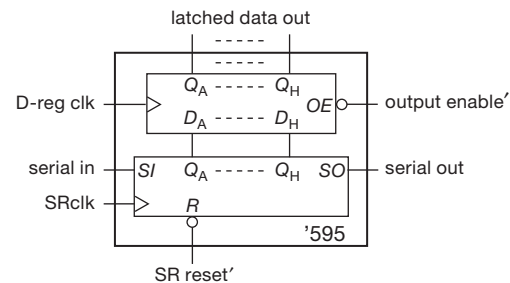


Figure 10.83. '595 octal shift register with output latch. A handy part for creating multiple latched outputs from a few pins of a microcontroller; the TPIC6595 variant has open-drain MOS drivers, good to 45 V and 250 mA, for driving heavy loads.

Inputs and outputs

Small shift registers can provide parallel inputs or outputs, and usually do; an example is the '395, a 4-bit parallel-in, parallel-out (PI/PO) shift register with three-state outputs. Larger registers may provide only *serial* input or output, i.e., only the input to the first flip-flop or the output from the last is accessible. In some cases a few selected intermediate taps are provided. One way to provide both parallel input and output in a small package is to share input and

⁶³ We could have ignored Q_A , and simply inverted Q_B (figure out why); however, for clarity we did full gating to detect $Q_A=0$ AND $Q_B=1$ for the synchronous reset signal SR'_1 .

⁶⁴ We mourn the passing of the unusual TIL306/7, a counter *with display* on one chip: you just looked at the IC, which lit up with a digit telling the count! Another fine example of an integrated counter is Intersil's ICL7216, an 8-digit 10 MHz universal counter-on-a-chip (complete with 7-segment LED driver); it's shown in all its glory in our second edition (on p. 526). The bad news is that it was recently discontinued; but the good news is that you can do it yourself, in an FPGA or cPLD – see Chapter 11. (Maxim still makes the ICM7217 4-digit up-down counter with multiplexed LED drive, and the ICM7218 and 7228 8-digit-LED drivers are available from both Maxim and Intersil.)

⁶⁵ There are even variable-length registers (e.g., the 4557: 1 to 64 stages, set by a 6-bit input).

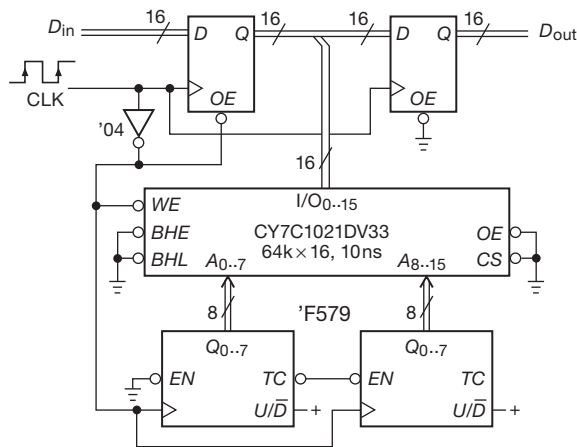


Figure 10.84. Large and wide shift register made from RAM plus counter; the slash indicates multiple lines, in this case 16-bit-wide data paths and a pair of 8-bit-wide addresses, for a total length of 65,536 16-bit words.

output (three-state) on the same pins, e.g., the '299, an 8-bit bidirectional PI/PO register in a 20-pin package. Some shift registers include a latch at the parallel input or output, so shifting can go on while data are being loaded or unloaded. A particularly nice example of the latter is the '595 8-bit shift register (Figure 10.83), nicely available in many logic families, including AHC(T), F, FCT, HC(T), LV, LVC, and VHC; it's great for creating a latched many-bit parallel output from a stingy bitstream coming from a microcontroller.⁶⁶ Similarly, the '597 is an 8-bit parallel-input shift register, convenient for getting data into a microcontroller via a single-bit serial input pin.

As with counters, parallel LOAD and CLEAR can be either synchronous or jam-load; for example, the '323 is the same as the '299, but with synchronous clear.

B. RAMs as shift registers

A random-access memory can always be used as a shift register (but not vice versa) by using an external counter to generate successive addresses. Figure 10.84 shows the idea. A pair of cascaded 8-bit synchronous counters generates successive addresses for a 64k-word $\times$ 16-bit static RAM. The combination behaves like a 16-bit wide and 65,536-long shift register. By choosing a fast counter⁶⁷

⁶⁶ The wonderful do-it-all microcontroller is the subject of Chapter 15, where we'll show an example.

⁶⁷ The '579 is available only in a 5 V logic family. The rest of the circuit is powered from +3.3 V; however, signal levels are compatible throughout (that is, the logic running on 3.3 V has "5V-tolerant" inputs; and the 5V logic accepts 3.3 V input levels; see §12.1.2A).

and memory, we were able to achieve a maximum clocking rate of 27 MHz (see timing diagram, Figure 10.85), which is comparable to that of an integrated (but much smaller) standard-logic shift register. (It's worth spending a few minutes studying that calculation – a good exercise in setup and hold times, propagation delays, and memory access timing.) This technique can be used to produce very large shift registers, if desired.

Exercise 10.27. In the circuit of Figure 10.84, input data seem to go into the same location that output data are read from. Nevertheless, the circuit behaves identically to a classic 64k-word shift register. Explain why.

10.5.4 Programmable logic devices

We've said it before (and we'll be saying it again) – contemporary digital design is moving relentlessly toward user-programmable ICs, typically containing from hundreds to hundreds of thousands(!) of gates and flip-flops,⁶⁸ in which the *connections* are programmable. Design entry is done in a hardware description language, processed by software to generate the connection *netlist*, then loaded into the chip via a serial interface (usually JTAG). We'll have more to say about these in the next chapter; but they do so many logic jobs so well that we cannot end this chapter without a short summary.

A. The bad news

To use these little puppies you usually need to learn an HDL like Verilog or VHDL, and you need a programming "pod" (or other link to the computer where the HDL is processed). And these devices are universally (well, almost) packaged only in surface-mount packages, which makes rough-and-ready prototyping more difficult.

B. The good news

You can preserve an ignorance of HDL programming by using instead schematic-entry alternatives that are available, both from the PLD manufacturers and from third-party suppliers (§11.3.3A). And for most digital jobs, PLDs really hit the spot. Here are the most important uses and advantages of PLDs.

State machines

The PLD is a natural for an arbitrary synchronous state machine. You would be foolish to use an array of *D* flip-flops and discrete combinational logic when a PLD does the job in one inexpensive and powerful package.

⁶⁸ Sometimes supplemented with dedicated functions such as RAM, interfaces, and processors.

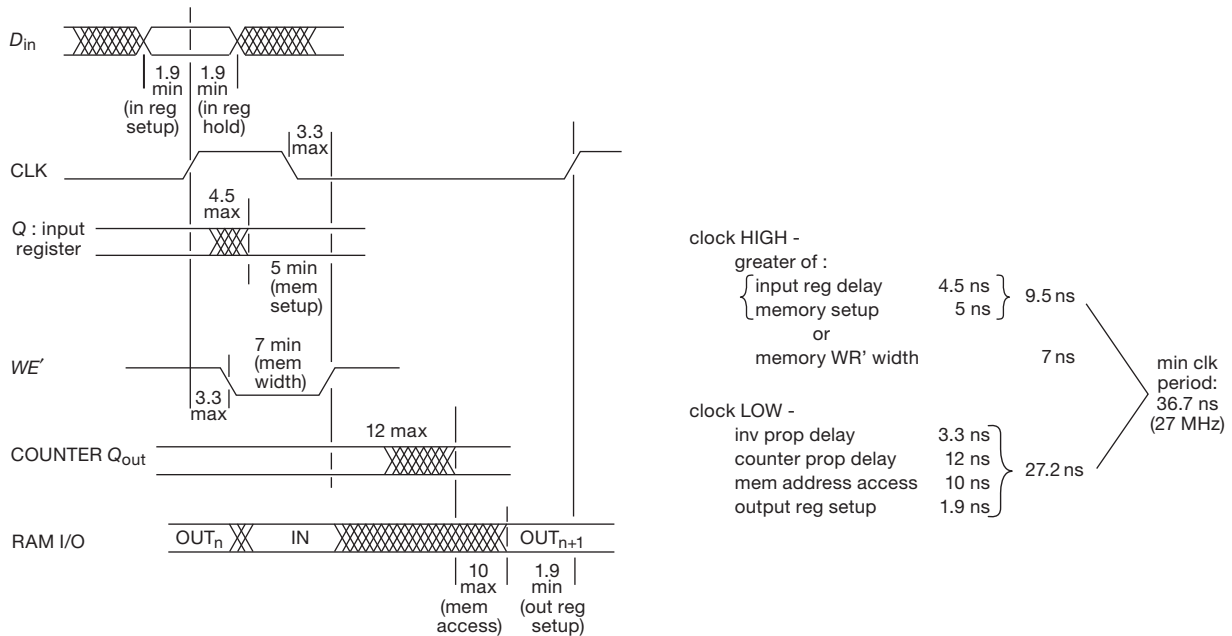


Figure 10.85. Timing diagram for the long shift-register implemented in RAM (the circuit of Figure 10.85), assuming worst-case timing specifications. This facilitates calculation of the maximum clock speed of 27 MHz.

Replacing “random” logic

Within many circuits you find little knots and tangles of gates, inverters, and flip-flops, scornfully referred to as *random logic*, or *glue logic*. A PLD will generally cut the package count by a factor of 10 or more.

Flexibility

Sometimes you’re not quite sure how you ultimately want some circuit to work, yet you must finish the design so that you can play around with it. PLDs are great here, because you can reprogram them at some later stage, without the rewiring you’d have to do if you had used discrete logic. With PLDs, the circuit itself is a form of software.⁶⁹

Multiple versions

PLDs make it possible to design a single circuit, then produce several different versions of the instrument by populating the board with differently programmed PLDs.

Speed and inventory

With PLDs you can generally get the design job done more quickly (once you’ve learned the ropes and set up your software tools). Furthermore, you need to stock only a few

PLD types, rather than dozens of standard-function MSI logic types.

System-on-a-chip

The larger PLDs (in particular, FPGAs) have enough resources so that you may be able to do your whole design in one PLD. In particular, you can include functions such as an interface (Ethernet, USB, or whatever), memory, and even a microprocessor on the single FPGA. There are two possibilities: (a) if you include such functions in your HDL design and let the software implement them within the array of gates and flip-flops, you’ve got a “soft” implementation; that may include nicely implemented functions that you get from elsewhere, in which case it’s called “intellectual property” (IP); (b) these functions may be hardwired into the FPGA (and not changeable) when you get it – that’s a “hard implementation.” More on this in the next chapter.

10.5.5 Miscellaneous sequential functions

With the relentless progress of the semiconductor industry, which routinely puts millions of transistors on a chip,⁷⁰ you

⁶⁹ More properly, *firmware*: in between unchangeable *hardware* and easily changed *software*.

⁷⁰ And *billions* in some of the larger microprocessors, graphic processors (GPUs), and FPGAs.

can get weird and wonderful gadgets all on one inexpensive chip. This brief subsection presents just a sampling.

First-in–first-out memory

A first-in–first-out (FIFO) memory is somewhat akin to a shift register in that data entered at the input appear at the output in the same order. The important difference is that with a shift register the data get pushed along as additional data are entered and clocked, but with a FIFO the data notionally *fall through* to the output queue. Input and output are controlled by separate clocks, and the FIFO keeps track of what data have been entered and what data have been removed.⁷¹

FIFOs are useful for buffering asynchronous data. The classic application is buffering a keyboard (or other input device, such as magnetic disk, or a fast external port such as Ethernet) to a computer or sluggish instrument. By this method, no data is lost if the computer isn't ready for each word as it is generated, provided the FIFO isn't allowed to fill up completely. The generic 7201–06 byte-wide series has been popular, and comes in several voltages and lengths; for example, 72V01–06 series of 3.3 V CMOS FIFOs has 0.5–16k words of 9 bits each, 40 MHz maximum speed, and zero *fall-through time* (an affliction suffered by early FIFOs, which were implemented as serial registers). You can get bidirectional FIFOs, synchronous FIFOs, and FIFOs up to 72 bits in width.

A FIFO is unnecessary if the device to which you are sending data can always get it before the next data arrive. In computer language, you must ensure that the maximum *latency* is less than the minimum time between data words. Note that a FIFO will not help if the data recipient is not able, *on the average*, to keep up with the incoming data.

Digital voltmeters

You can get complete digital voltmeters (DVMs) on a single chip (for some reason you'll find these things listed as "display drivers" when you go searching at distributors). They include voltage reference, high-impedance differential inputs, LCD drivers, and so on. An example is the MAX1495, a fully integrated $4\frac{1}{2}$ -digit voltmeter in a 7 mm square package; it draws about a milliamp from a single 3–5 V supply. A popular generic part is the '7135 (TI, Intersil, and Maxim call it ICL7135; Microchip calls theirs TC7135; and TI also sells it as the TLC7135): it's a $4\frac{1}{2}$ -digit voltmeter that runs from +5 V and drives 7-segment

LED displays. The '7136 is $3\frac{1}{2}$ -digit and drives 7-segment LCDs.

Special-purpose circuits

There are nice collections of large-scale integration (LSI) chips for arcane jobs like radio communications (e.g., frequency synthesizers), digital signal processing (digital filters, correlators, Fourier transforms, arithmetic units), data communications (UARTs, modems, network interfaces, data encryption–decryption ICs, serial format converters, wireless protocols), and the like. Often these chips are used in conjunction with microprocessor-based devices, and many of them cannot stand alone.

Consumer chips

The semiconductor industry loves to develop ICs for use in large-market consumer products. You can get single chips to make digital (or "analog") watches, clocks, locks, calculators, smoke detectors, telephone dialers, music synthesizers, rhythm and accompaniment generators, etc. The guts of radios, TVs, music and video players, GPS navigators, and cellphones are nearly empty these days, thanks to large-scale integration. Speech synthesis and recognition is well developed: that's why devices like GPS navigators can speak to us and understand our analog-generated replies.⁷² Automobiles are laden with dozens of processors, for tasks like engine control, braking, collision-avoidance systems, navigation, and so on. Even the lowly toothbrush has a processor chip, running on a few thousand lines of computer code.⁷³

Microprocessors

The most stunning example of the wonders of very large-scale integration (VLSI) is the microprocessor, a computer on a chip. At one extreme there are powerful number crunchers like the Intel eight-core Itanium, with more than 3 billion transistors; it has hundreds of internal registers, supports up to a petabyte (a million gigabytes) of RAM, and can be cobbled together into 512-processor architectures. At the other extreme are inexpensive single-chip processors with a rich load of input, output, and memory functions included on the same chip for stand-alone use.

⁷² One of the authors' children, enchanted with the talking Nissan Maxima we rented one summer, augmented its lexicon, which included messages such as "door is a-jar" and "tank is low," with his own: "You ran into a dinosaur."

⁷³ Check out the MC9RS08KA from Freescale Semiconductor, with internal 10 MHz oscillator and 2 KB of internal memory; it's "small enough to fit in the head of an electric toothbrush" and intended for "personal care appliances," among other uses. It costs a mere \$0.40 in quantity.

⁷¹ A FIFO can be implemented in software (as it usually is done in hardware) by creating a *ring buffer* in RAM, with a pair of pointers (write and read).

An example (Figure 10.86) is the ARM7 LPC2458 from NXP (formerly Philips): 512kB/64kB memories, 72 MHz clock, 10/100 Ethernet, USB 2.0, 10-bit A/D and D/A, 2xPWM, 4 UARTs, 2xCAN, SPI, 2xSSP, 3xI²C, I²S, 136 bits of general-purpose I/O, and an external memory controller. This puppy costs just \$10!⁷⁴ This latter type is intended as a dedicated controller in an instrument rather than as a versatile computation device.

The microprocessor revolution hasn't begun to slow, and we have seen a doubling of computer power and memory size (now 8 Gbit per chip, compared with 1 Mbit/chip and 16 kbit/chip at the time the two previous editions of this book were written) every 18 months ("Moore's law"); at the same time, prices have dropped dramatically (Figure 10.87). Along with bigger and better processors and memory, recent activity in display devices, networking, and wireless data communication promises yet more excitement in coming years.

10.6 Some typical digital circuits

Thanks to the efforts of the semiconductor industry, digital design is wonderfully easy and pleasant. It's almost never necessary to "breadboard" a digital circuit, as so often is the case with analog design. Generally speaking, the only serious pitfalls involve timing and noise. We'll have more to say about the latter in Chapter 12.

This is a good place to illustrate timing with some sequential design examples. Some of these functions can be performed with LSI circuits or with programmable logic, but the implementations shown are reasonably efficient, and illustrate the kind of circuit design that's straightforward with widely available parts (and without mastery of any software languages or tools).

10.6.1 Modulo- n counter: a timing example

The circuit in Figure 10.88 produces one output pulse for every $n+1$ input clock pulses, where n is the 8-bit number you've set on the pair of hexadecimal thumbwheel switches. The '163s are 4-bit synchronous up-counters, with synchronous load (when LD' is LOW) via the P_n inputs. The idea is to load the counters with the *complement*

of the desired count, then count up to FF_H , reloading at the next clock pulse. Because we've generated the preload levels with pullups to the positive supply (with the switch common grounded), those levels are LOW-true for the displayed switch settings; that makes the preload values, interpreted as active-HIGH, equal to the 1's complement of the switch settings.

Exercise 10.28. Show that the last statement is true by figuring out the active-HIGH value that will be loaded for the switch settings in Figure 10.88.

Circuit operation is entirely straightforward. To cascade synchronous counters, you tie all clocks together, then tie a "maximum-count" output of each counter to an enable of the successive counter. For an enabled '163, the RCO (ripple-clock output) goes HIGH at maximum count, enabling the second counter via the enabling inputs ENT and ENP. Thus IC₁ advances at each clock, and IC₂ advances at the clock after IC₁ reaches F_H . The pair thus counts in binary until the state FF_H , at which point the LD' input is asserted. This causes synchronous preload at the next clock. In this example we've chosen counters with *synchronous* load to avoid the logic race (and runt-pulse RCO) that you would get with a *jam*-loaded counter. Unfortunately, this makes the counter divide by $n+1$ rather than by n .

Exercise 10.29. Explain what would happen if jam-load counters (e.g. '191s) were substituted for the synchronous-load '163s. In particular, show how a runt pulse would be created. Demonstrate also that the foregoing circuit divides by $n+1$, whereas the asynchronous-load version would divide by n (if it worked at all!).

A. Timing

How fast can our modulo- n counter count? Let's use LV-family logic at 3.3 V, at which the 74LV163A specifies a guaranteed $f_{\max}$ of 70 MHz.⁷⁵ However, in our circuit there are additional time delays associated with the cascading connection (IC₂ has to know that IC₁ has reached maximum count in time for the next clock pulse) and also the load-on-overflow connection. To figure the maximum frequency at which the circuit is guaranteed to work, we have to add up the worst-case delays and make sure there is enough setup time remaining. Look at Figure 10.89, where we've drawn a timing diagram showing the load sequence that occurs at maximum count.

⁷⁴ NXP's LPC1768 is a low-power part that's popular with the prototyping crowd. It has most of the features of the LPC2458, but lacks the external memory controller. It has only 70 GPIO pins (we're sure you'll not miss the rest), and it comes in an easy-to-solder 100-pin QFP package (0.5 mm pin spacing). Notably it has a 12-bit ADC, a 10-bit DAC, and six general-purpose PWM outputs.

⁷⁵ The '1G04 isn't available in LV, so we'll use a 74LVC1G04.

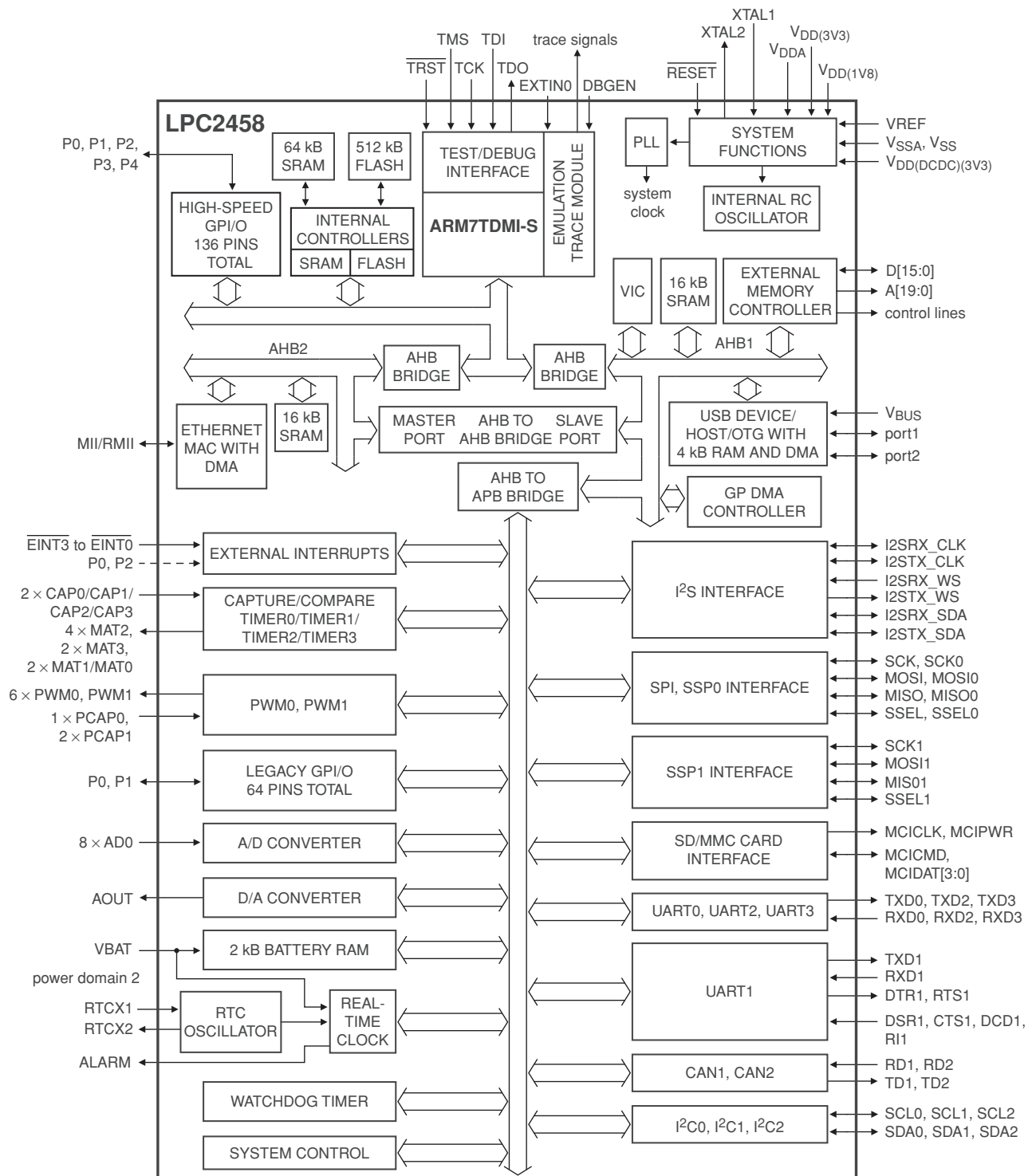


Figure 10.86. A preview of Chapter 13: an inexpensive microcontroller festooned with lots of cool stuff. Adapted from Document LPC2131_32_34_36_38_4 ©NXP B.V. 2007)

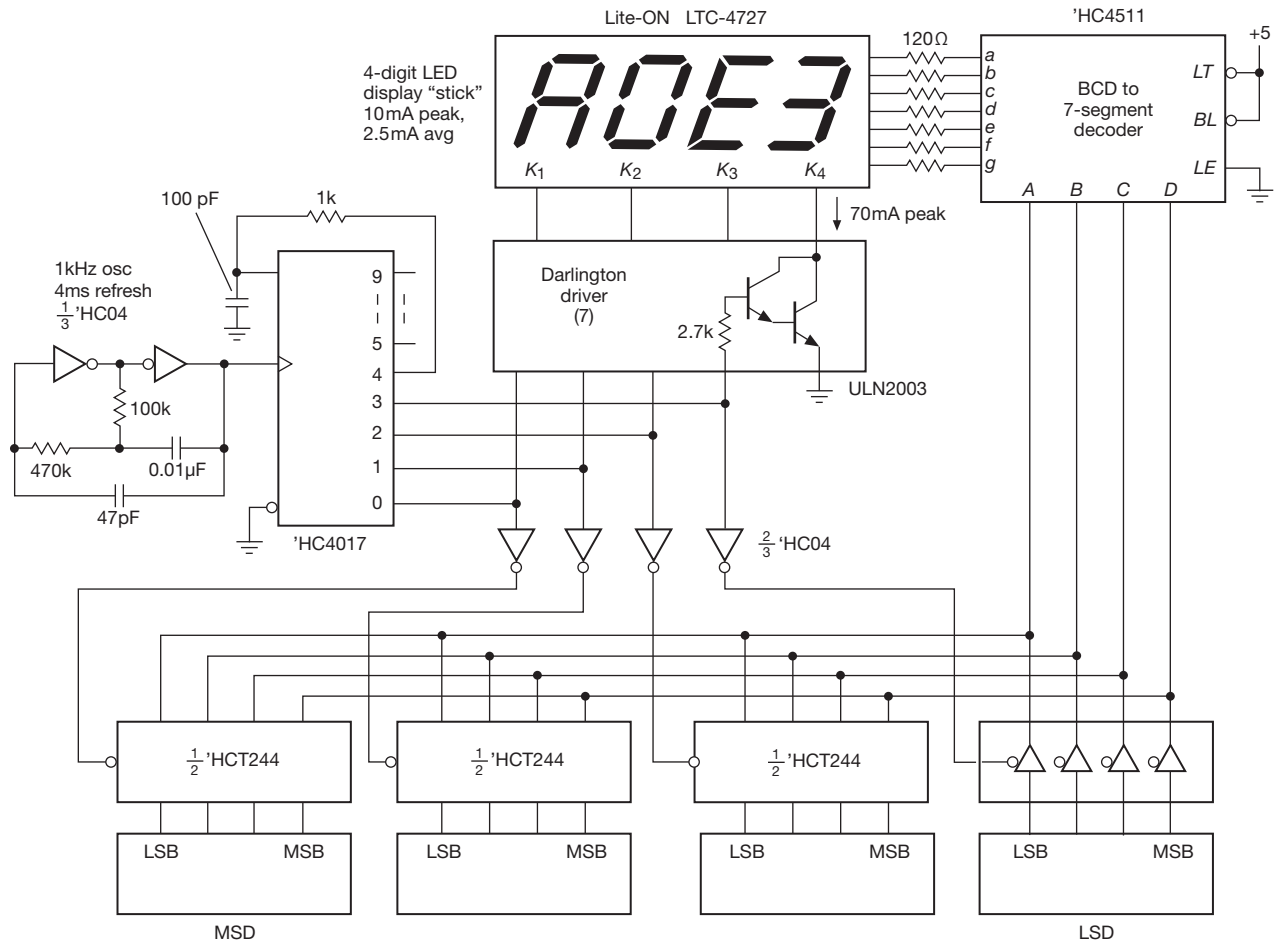


Figure 10.90. Four-digit multiplexed display. Whatever might that displayed quantity represent?

parallel load (synchronous *or* jam-load!), decoded zero-state output, and reset-to-maximum input.

10.6.2 Multiplexed LED digital display

This example illustrates the technique of display multiplexing: displaying an n -digit number by displaying successive digits rapidly on successive 7-segment LED displays (of course, the characters need not be numbers, and the displays can have a different organization than the popular 7-segment arrangement). Display multiplexing is done for reasons of economy and simplicity: displaying each digit continuously requires separate decoders, drivers, and current-limiting resistors for each digit, as well as separate connections from each register to its corresponding decoder (4 lines) and from each driver to its corresponding display (7 wires); it's a mess!

With multiplexing, there's only one decoder/driver and one set of current-limiting resistors. Furthermore, because LED displays come in n -character "sticks" with the corresponding segments of all characters tied together, the number of interconnections is greatly reduced. An 8-digit display requires 15 connections when multiplexed (7-segment inputs, common to all digits, plus one cathode or anode return for each digit), rather than the 57 that would be required for continuous display (and, since most LED displays come in multiplexed varieties, you'll probably find them to be the best choice anyway).

Figure 10.90 shows the schematic diagram. The digits to be displayed are resident in registers at the bottom; they could be counters (e.g., if the device happened to be a frequency counter), or perhaps a set of latches receiving data from a computer, or possibly the output of an ADC, etc. In any case, the technique is to assert each digit successively

onto an internal 4-bit “bus” (in this case with four 4-bit three-state buffers, each half of an ’HCT244 octal buffer), and decode and display it while asserted on the bus (using the ’HC4511 BCD-to-7-segment decoder/driver).

We’ve used a pair of inverters to form a classic CMOS oscillator (Figure 7.2) operating at about 1 kHz, driving a ’HC4017 decimal counter/decoder. As each successive output of the counter goes HIGH, it enables one digit onto the bus and simultaneously pulls the corresponding digit’s cathode LOW via the popular ULN2003 high-current open-collector Darlington driver. The ’HC4017 is rigged up to cycle through the states 0–3 by resetting when the count reaches 4 (with a bit of RC delay to ensure full reset). Display multiplexing will work with greater numbers of digits, and it is universally used in instruments with multidigit LED displays. Try rapidly shifting your eyes while viewing such a display – you get numeric alphabet soup!

One of the interesting design issues has to do with the LED “voltage budget”: you’ve got a +5 V supply, and the LEDs that make up the segments have a voltage drop of ~ 2.2 V at the desired 10 mA drive current, so it would seem that there’s nothing to worry about. A closer look (Figure 10.91) shows the problem, namely that the ’HC4511 anode drivers (which are a pMOS–npn hybrid) have one V_{BE} drop (about 0.7 V); and the ULN2003 Darlington digit sinks, running at 70 mA, have another V_{BE} plus some saturation voltage, for a total of about 0.9 V. Add the 2.2 V LED drop,⁷⁶ and you’ve got only 1.2 V left across the segment current-limiting resistors. So, for 10 mA LED current we needed 120 Ω resistors. That’s pretty much OK, although you’d be forgiven for worrying that some scatter in the LED forward drops, etc., would produce significant percentage changes in the small remaining voltage across the resistors; i.e., you might see unacceptable brightness variations across the digits. If you’re still not worried, consider the effect of a 10% drop in the +5 V supply. Note, also, that this LED drive circuit would not work at all with a +3.3 V supply.⁷⁷

In an LED the maximum allowable current is limited only by overheating. So it’s OK to use rather high peak currents in a multiplexed display, as long as the average current stays within the rated value. You have to be a bit careful, though: the thermal time constant of the small LED semiconductor chip is somewhere down around a millisecond, so for significantly longer ON times the peak cur-

rent cannot much exceed the rated maximum average current. Another fine way to destroy a pulsed LED is to have the drive circuit get stuck in the ON state, for example caused by a program crash while debugging a microcontroller (firmware-driven) multiplexed display.⁷⁸

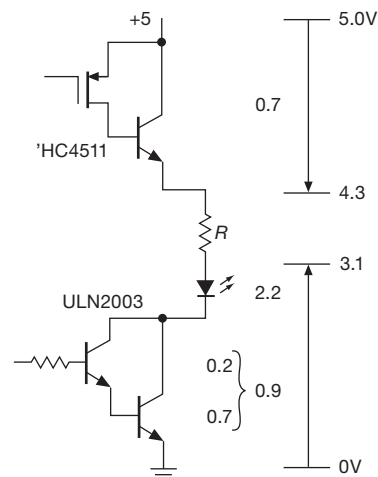


Figure 10.91. LED voltage budget.

Many LSI display-oriented chips, included in devices such as clocks, set-top (television) boxes, and the like, include on-chip display multiplexing (and even driver) circuitry. You can also get (though with some difficulty⁷⁹) a stand-alone 6-digit display driver-multiplexer, the legacy 74C912; it accepts the 4-bit characters in sequence, and requires only the external digit driver.

10.6.3 An n -pulse generator

The n -pulse generator is a useful little test instrument. It generates a burst of n output pulses following an input trigger signal (or you can push a button), with a set of selectable pulse repetition rates. Figure 10.92 shows the circuit.⁸⁰ The ’HC190s are decade up/down counters (here wired to count down), clocked continuously by a selected power-of-10 subdivision of the fixed 10 MHz crystal oscillator, but disabled by having both the ALD’ (asynchronous load) inputs asserted and the EN’ (count enable) inputs dis-

⁷⁶ As specified for the yellow display stick shown in the figure; the forward drop of an LED depends on color; see Figure 2.8.

⁷⁷ Nor with a 5V supply and blue LED display, where the forward voltage $V_f \sim 3.5$ V.

⁷⁸ See §7.2.3 for a nice way to prevent this particular disaster.

⁷⁹ Jameco shows them in stock, \$2 in unit quantity.

⁸⁰ Challenge for the reader: what is so “perfect” about the figure?

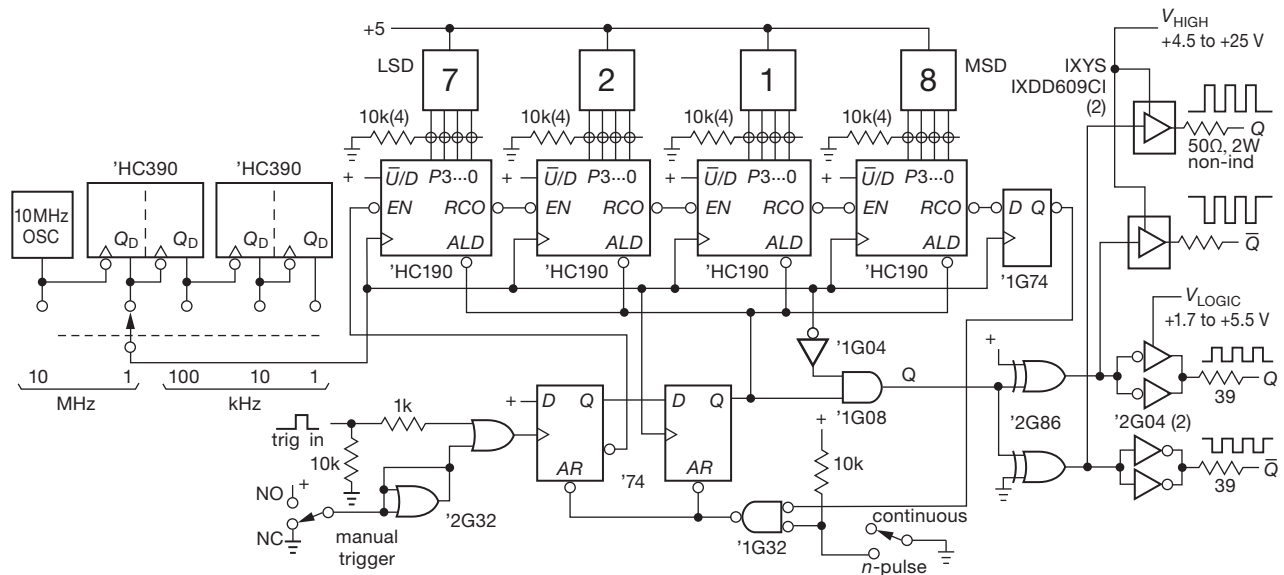


Figure 10.92. Perfect n -pulse generator. All logic is LVC family, running from +5 V, unless shown otherwise.

Pulses (during clock LOW) are passed by the AND gate until the counter reaches zero, at which state the RCO' output⁸¹ is latched for one clock, and both flip-flops are reset; this parallel-loads the counter back to $n-1$ from the BCD switches, disables counting, and readies the circuit for another trigger.⁸² Note that the use of pulldown resistors in this circuit means that true (rather than complemented) BCD switches must be used. Note also that the manual trigger input must be debounced, as shown, since it clocks a flip-flop. That is not necessary for the continuous/ n -pulse switch, which simply enables a continuous stream of output pulses.

The output stage delivers two pairs of true/complement signals: the XORs generate complementary logic signals with equal delay; the paralleled 'LVC2G04 inverters give

normal rail-to-rail logic swings, settable from +1.7–5.5 V via an external dc supply input. We used two inverters in parallel to increase drive capability (the paralleled inverter sections can sink or source 32 mA, staying within 0.5 V of the rails, for $V_+ = 3$ V); the 39 Ω resistors, in combination with the $\sim 10 \Omega$ inverter-stage output impedance, provide series termination for 50 Ω cable.

We added the boxed driver pair for serious driver tasks. It uses a hefty “MOSFET driver” chip, intended for rapid switching of highly capacitive MOSFET gate inputs. This particular specimen can sink or source up to 8 A, with a better-than-average switching time of ~ 10 ns; it is noninverting, and accepts standard 5 V logic swings. The outputs are series terminated, with non-inductive 50 Ω 2 W resistors.⁸³

⁸¹ Indicating a ripple-through “terminal count,” which for a down-counter (as here) is the 4-digit state 0000. A peculiarity of the '190 is that RCO' is enabled only during clock LOW.

⁸² Because of the RCO' latch, the output pulse train has $n+1$ pulses (rather than n), so you have to set the switches to one fewer than you want. Omitting that latch, however, would trade this *numerical* embarrassment for a *logical* one, namely the generation of a terminal output runt pulse. Challenge yourself by figuring out why. However, further removing the 1G04 clock inversion would eliminate the output runt, producing instead a much-shortened reload pulse, while retaining the $n+1$ counting “feature.” There's nothing wrong, really, with a reload pulse whose width is determined only by logic propagation delays – it's just a bit, well, *ugly*; by contrast, the latched RCO' for asynchronous reset-reload is neat and clean (and easy to see on a 'scope).

10.7 Micropower digital design

Small battery-powered gadgets of all sorts need to operate on very small currents, ideally down in the microamp range. To appreciate the scale, consider that a 9 V battery has a capacity of about 500 mAh, thus about 20 days at 1 mA current drain; and a small “coin cell” like the

⁸³ Readers interested in constructing this circuit should consider, as an alternative to the '190 (or '192) counters, the elegant 'HC4059 4-decade loadable down-counter, which replaces four chips with one. If, however, *hexadecimal* counting is to your taste (perhaps you have 16 fingers?), use the 'HC191 (binary) alternative.

ever-popular CR2032 (stock check: nearly two million in stock at DigiKey today) delivers about 200 mAh at 3 V.

There are lots of micropower chips out there – both linear (op-amps, voltage references, oscillators, etc.) and digital (standard and programmable logic, ADCs and DACs, microcontrollers, etc.) – which is where any micropower design should start.

But if you're not careful it's easy to sacrifice the micropower performance of these excellent chips. The previous edition of this book⁸⁴ has a full chapter (Chapter 14) on micropower design. It includes lots of information on batteries and other power sources, and on both linear and digital micropower design; it makes good reading. But for reasons of space we are limited here to a brief subset of this important area, namely the business of micropower digital logic.

10.7.1 Keeping CMOS low power

There are several routine measures you should take to achieve low-current CMOS operation. In addition, it's worthwhile raising your CMOS pathology awareness.

A. Routine design considerations

- *Keep as few nodes as possible involved with high frequencies.* CMOS has no quiescent current (other than leakage), but current is required to charge internal (and load) capacitances during switching. Since the energy stored in a capacitor is $\frac{1}{2}CV^2$, and an equal amount of energy is dissipated by the resistive charging circuit, the power dissipated is

$$P = V_{DD}^2 f C$$

for a switching frequency f . Thus CMOS devices consume power proportional to their switching frequency, as we saw in Figure 10.27. At their maximum operating frequency they may use more power than equivalent bipolar-TTL logic. The effective capacitance C is often given on datasheets as the “power-dissipation capacitance,” C_{pd} , to which you must add the load capacitance C_L before applying the formula above.

- *Within a circuit, be careful whenever mixing supply voltages.* Otherwise you may have current flowing through input-protection diodes. Even worse, you may force a chip into SCR latching (see §10.8.3B).
- *Make sure logic swings go all the way to the rails.* CMOS outputs swing rail-to-rail. Outputs from other devices – bipolar TTL, oscillators, NMOS chips – may hover in

between, causing class-A current and decreased noise immunity (recall Figure 3.93).

- *No open inputs.* Open inputs are the enemy of micropower operation, since there may be considerable class-A current (and even oscillation) as the input floats to the logic threshold (Figure 10.101). Tie all unused inputs to ground (or V_{DD} , if that disables something you don't want).
- *Arrange loads to keep normal-state drains low.* Pullups, pulldowns, LEDs, and output drivers should be wired so that current is minimum in the usual state.
- *Avoid slow transitions.* Again, class-A current is the culprit. A sinewave input driving a CMOS Schmitt trigger may cause a lot of supply current.
- *Put current-sensing resistors in the V_{DD} lead.* In certain failure modes, particularly those caused by static damage, a CMOS chip may draw excessive quiescent current; a $10\ \Omega$ resistor in series with V_{DD} on each board (Figure 10.93) makes it easy to see if that is happening (and you can use the stuck-node tracer in §4.8.2 to track down the offending component).

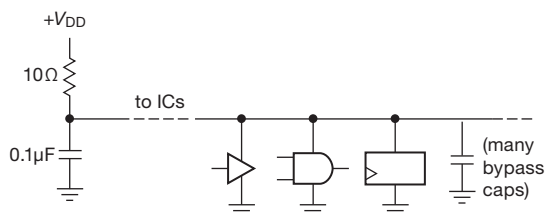


Figure 10.93. A “current spy” makes it easy to locate circuit subsections wherein lurks a current-hogging IC.

- *Quiescent current screening.* A typical CMOS logic chip (in any family – 4000B, HC, LVC, LCX, AUC, etc.) has a specified I_Q of 5–20 μA *maximum*, but down around 0.04 μA *typical* (if specified at all). Manufacturers evidently set a conservative spec on maximum leakage, probably because they don't want to bother testing to a much lower realistic value. Most of the time it is rare to have a quiescent current anywhere near the maximum, but it can happen. If you are operating at low switching frequencies (therefore low dynamic current), and require comparably low quiescent current, you may need to screen incoming chips. The use of small series resistors as recommended above makes the job much easier. We've noticed that in the case of CMOS LSI chips (such as large memories) the typical quiescent current may be close to the manufacturer's maximum leakage specifications – beware!
- *Time-out power switching.* You can save a lot of power by

⁸⁴ With continued (and perhaps perpetual) availability as an e-book.

making sure an instrument is turned off when no one is using it. Back in Chapter 7 we showed a simple time-out circuit, built with discrete logic, that turns off the switched +9 volt power an hour after the instrument has been turned on (§7.2.4A). Better yet, in any instrument with an embedded microcontroller (Chapter 15), use the controller's internal timer (or a programmed time-out loop) to command the power switching. In applications with limited battery energy, it's best to choose a micropower controller; alternatively, arrange for it to spend most of its time in a low-power mode (variously called "idle," "power-save," "power-down," "standby," "hibernate," or "sleep").

10.8 Logic pathology

There are interesting, and sometimes amusing, pitfalls awaiting the unsuspecting digital logician. Some of these, such as logic races and lockup conditions, can occur regardless of the logic family in use. Others (e.g., "SCR latchup" in CMOS chips) are genetic abnormalities of one logic family or another. In the following subsections we have collected our bad experiences in the hope that such anecdotes can help others avoid such problems.

10.8.1 dc problems

A. Lockup

It is easy to fall into the trap of designing a circuit with a lockup state. Suppose you have some gadget with a number of flip-flops, all going through their proper states. Everything seems to work fine. Then one day it just stops dead. The only way you can get it to work is to turn the power off and back on again. The problem is that there is a lockup state (an excluded state of the system that you can't escape from), and you got into it because of some powerline transient that sent the system into the forbidden state. It is important to look for such states when you design the circuit, and rig up logic so that the circuit recovers automatically. At a minimum, things should be arranged so that a RESET signal (generated manually, at startup, etc.) brings the system to a good state. This may not require any additional components (e.g., Exercise 10.23).

B. Startup clear

A related issue is the state of the system at startup. It is always a good idea to provide some sort of RESET signal at startup. Otherwise the system may do weird things when first turned on. One approach is to use an RC charging waveform, buffered by a Schmitt trigger (Figure 10.94).

In addition to requiring several discrete components, however, this circuit has the drawback of not reliably responding to a momentary voltage dip.

A better approach is provided by a power-supply *supervisory* circuit IC. These chips come in many varieties. The simplest are 3-pin parts that create a reset pulse on power-up, for example, the venerable MC34164, which comes in a convenient TO-92 leaded transistor package (in addition to the usual tiny surface-mount configurations), and holds its open-collector output LOW until the supply voltage rises above 4.3 V (2.7 V for the 34164-3 part); it includes an internal voltage reference, and some hysteresis. A more flexible part is typified by the Maxim MAX700, which comes in 8-pin packages (including DIP), and which provides both RESET and RESET' active pullup outputs; it lets you set the threshold in the range of 1.2–4.7 V with external resistors (you can also set the hysteresis), and it has an input for a manual RESET switch (Figure 10.95). The MAX823 and ADM823 families are "jellybeans" that are widely second-sourced. Other supervisory chips include the so-called *watchdog* function: You have to pulse it at least once per second, or it does a reset; these are intended to sense a processor crash, and force a reboot (a function often integrated into contemporary microcontrollers, as well).

Many of these supervisory parts are made by multiple

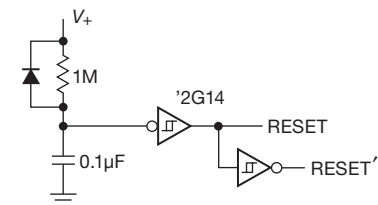


Figure 10.94. Simple power-on reset.

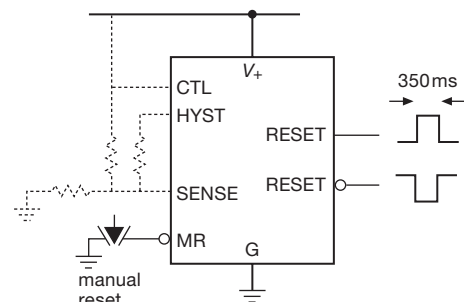


Figure 10.95. Supervisory IC provides power-on, voltage monitor, and manual reset functions. The optional (dotted) circuitry permits threshold and hysteresis adjustment.

manufacturers, but with different prefix part names. For example, the MAX809 is also available with APX, ADM, CAT, LM, STM, and TCM prefixes, and costs \$0.14 in a roll of 3,000 pieces, thanks to all the competition. But be careful and read the datasheet before making a substitution! For example, NSC’s MCP809 datasheet says it has a different pinout from the other ’809 parts (it matches STM’s jellybean STM1001). A drafting error? Maybe, but we warn you, be careful.

Advanced parts like the ADM690 series add features like supply switching to a backup battery, a second low-voltage warning comparator, and chip-enable gating. See Table 10.6 for characteristics of some favorite supervisory ICs.

Table 10.6 Selected Reset/Supervisors

Type	Pins	Package(s)	# Voltages ^a	Watchdog	Reset IN	Active High ^c	Supply current (μA)
MC34164	3	TO-92, SO-8	2	-	-	-	12
MAX809	3	SOT-23	7	-	-	'810	15
NCP303	5	SOT-23	7	-	-	'302	0.5
MAX700	8	SOIC-8	1 ^b	-	•	both	100
ADM811	4	SOT-143	6	-	•	'812	5
ADM823	5	SOT-23, SC-70	7	•	•	'824	10

Notes: (a) Voltage is specified by a suffix. (b) External adjustment. (c) All have active-LOW output; part numbers listed have active-HIGH output instead; MAX700 has both.

10.8.2 Switching problems

A. Logic races

Lots of subtle traps lurk here. The classic race was illustrated with the pulse synchronizer in §10.4.4. Basically, in any situation where gates are enabled by signals coming from flip-flops (or any clocked device), you must be sure that a gate doesn’t get enabled and then immediately disabled after a logic delay time. Likewise, make sure that signals appearing at flip-flop inputs aren’t delayed with respect to the clock (another plus for synchronous systems!). In general, delay the clock rather than the data.

It is surprisingly easy to overlook a race condition. A classic example is the humble 2-input multiplexer: if you do the obvious thing, namely $Y = S \& A | \sim S \& B$, as in Figure 10.96A, you are in trouble! The problem occurs when there is a HIGH level on both *A* and *B* inputs, and the SELECT input *S* transitions from HIGH to LOW, i.e., from se-

lecting *A* to selecting *B*. The inverter’s delay causes the lower AND gate to be disabled before the upper AND is enabled, producing a transient LOW glitch at the output. The solution is to add the redundant term $A \& B$, as drawn in Figure 10.96B. (This is done correctly in any commercial IC. The problem comes when you want to put a MUX into programmable logic,⁸⁵ for which you spell out the logic in a hardware description language. You have to instruct it, sternly, not to “optimize” away the redundant term. You have to tell it that you are wiser than it. You can tell it, if all else fails, that you’ve read this book.)

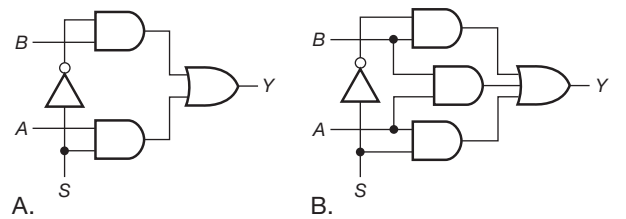


Figure 10.96. Adding a redundant term eliminates a logic-race glitch in the 2-input MUX.

B. Metastable states

As we discussed in detail earlier (§10.4.2D), a flip-flop (or any clocked device) can get confused if the data input changes during the setup-time interval preceding a clock pulse. As long as the flip-flop makes *some* decision promptly in this ambiguous case, all is well. However, there is a chance that the input may have changed at just the wrong time, at exactly the “moment of truth,” such that the flip-flop can’t make up its mind; its output can hover at the logic threshold for many times the usual propagation time (or it may sit in one logic state, then change its mind later, as in Figure 10.60).

This problem does not arise in properly designed synchronous systems in which setup times are always satisfied (by using logic fast enough so that inputs to flip-flops are stable by t_{setup} before the next clocking pulse). However, it can create problems in situations where asynchronous signals (e.g., going from device A, with its own clock, to device B, with a separate clock) must be synchronized. In these cases you cannot guarantee that input transitions do not occur during the setup interval; in fact, you can calculate how often they do!⁸⁶ The metastability problem has

⁸⁵ The subject of the next chapter.
⁸⁶ The chance of landing in the “setup interval” Δt of the faster clock (from t_{su} before the clock to t_{h} after the clock) is $\Delta t / t_{\text{clkF}}$, where $t_{\text{clkF}} = 1 / f_{\text{clkF}}$ is the period of the faster of the pair of asynchronous

been blamed for mysterious computer crashes, although we are skeptical. The cure generally involves a set of concatenated synchronizers, or a “metastable state detector” that resets the flip-flop. You may find claims of “metastable-resistant” logic, for example the AC(T) family of 5 V logic.

C. Clock skew

Clock skew problems arise when you have a clocking signal of slow rise time driving several interconnected devices (Figure 10.97). In this case two ’595 octal shift registers have been cascaded to create a 16-bit parallel latched output; they are being clocked by a slowly rising edge, caused by capacitive loading of an anemic clocking signal (perhaps coming from a slow microcontroller’s output pin). The problem is that the first register may have its threshold at a lower voltage than that of the second register (owing to process variations), and this causes it to shift earlier than the second register. The last bit of the first register is then lost. CMOS devices can display quite a spread of input threshold voltages, which compounds the problem (the threshold specification may be one-third to two-thirds of V_{DD} , and they mean it!). The best cure in such a situation is to drive the clock inputs from a nearby chip of adequate speed and without excessive capacitive loading. (Another way to fix the problem, if you’ve got clock skew, is to add some small delay in the data lines between successive clocked chips; but don’t let that substitute for a clean clock.)

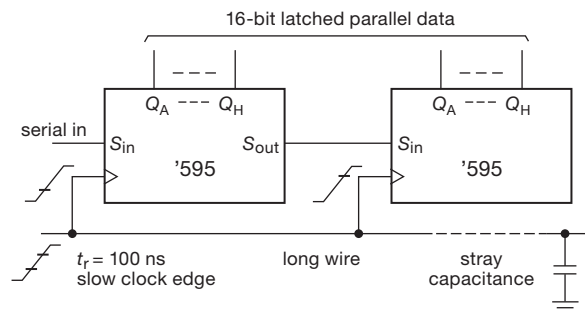


Figure 10.97. Slow risetimes can cause clocking skew, when the thresholds differ.

Speaking generally, edge-triggered clock inputs on any digital IC should always be treated with respect. For example, clock lines with noise or ringing should always be cleaned up with a gate (perhaps one with input hysteresis) before driving the clocked chip (but be careful not to vio-

late setup- and hold-time requirements). You’re especially likely to have problems with clock lines that come from another board, or from a different family of logic. For example, slow 4000B or 74C logic driving the faster HC or AC families is likely to exhibit problems of clock skew or multiple transitions; ditto for HC driving LVC, and so on.

Clock-skew problems can occur, surprisingly, even within a single chip of programmable logic. An example we’ve encountered is the venerable 9500-series of cPLDs, in which individual flip-flops can be clocked either by one of the chip’s distributed *global clock* signals, or, alternatively, by the output of internal logic (these are called *product terms*). Sounds fine. But if you use a product term to clock a set of flip-flops in a shift register, say, the chip may malfunction. That is because of routing delays in bringing that clocking signal to the several flip-flops. A synchronous circuit like this is only guaranteed to work if clocked by a global clock.⁸⁷ This “gotcha” is not prominently featured on the datasheets.

D. Runt pulses

In §10.6.1 (modulo- n counter) we used synchronous-load counters (’163), rather than a jam-load alternative (like the ’191), because with the latter you would need to add some delay to prevent a pulse of substandard width (since the counter’s output causes it to clear itself). The same comment goes for LOAD pulses when you are using counters or shift registers. Runt pulses will make your life miserable, because you may have marginal operation or intermittent failures. Use the worst-case propagation delay specifications when designing.

E. Unspecified rules

As the semiconductor industry was finding its way, beginning with the simplest RTL ICs of the 1960s (see §12.1.1 for a brief chronology), then the improved TTL and Schotky families, to the modern high-performance CMOS families, there was an understandable lack of standardization of pinouts, specifications, and functionality. As examples, the 7400 (NAND) had its gates pointing “down,” but the 7401 (open-collector NOR) was built with the gates going the other way. This created so much confusion that it had to be mutated into the 7403, which is a 7401 with 7400-style pinouts; a similar disaster happened with the 7490 (BCD ripple counter), with power-supply pins in the middle instead of at the corners. (Ironically, mid-chip power-

clocks. Meanwhile, the slower clock is ticking along at f_{CLKS} . So it lands in the metastable interval, on average, at a rate of $f_{\text{CLKS}}f_{\text{CLKF}}\Delta t$ per second.

⁸⁷ Which, unfortunately, you cannot drive from a product term, except by bringing the signal out on one pin and back in (to a global clock) on another.

supply pins have made a comeback in fast CMOS, because of their reduced inductance.)

An important legacy of this early anarchy is the hodgepodge of “unspecified rules” that we’re stuck with. For example, the ever-popular ’74 type-*D* flip-flop exists in every logic family; asserting both SET and CLEAR makes both outputs HIGH in every family except 74C, where it makes both outputs LOW! That’s not exactly an unspecified rule, since if you look carefully in the fine print you’ll find the inconsistency; the technical term for it is a *gotcha*. Another of our favorite gotchas is the ’96, a 5-bit shift register with tricky jam-load inputs: they can SET, but they cannot CLEAR!

A genuine unspecified rule, and in fact a very important one, is *removal time*: that’s the amount of time you must wait after disasserting a jam-type input before a clocked device is guaranteed to clock properly. Chip designers didn’t bother specifying this until the logic families of the early 1980s (though circuit designers always wanted to know it), specifically the advanced Schottky and fast CMOS families. If you’re designing with earlier logic, our advice is to be conservative; for example, assume that the removal time is the same as the data setup time.⁸⁸

10.8.3 Congenital weaknesses of TTL and CMOS

We divide this section into nuisance problems and really bizarre behavior.

A. Nuisance problems

Bipolar (legacy) TTL

You have to remember that TTL inputs *source* current when held in the LOW state (e.g., 0.25 mA for LS, 0.5 mA for F). That makes it difficult to use *RC* delays, etc., because of the low impedances necessary, and in general you have to give some thought when interfacing linear levels to TTL inputs.

The TTL threshold (and that of its imitators, HCT and ACT) is too close to ground, making the whole logic family somewhat noise prone (more on this in Chapter 12). The high speed of these logic families makes them recognize short spikes on the ground line; those spikes, in turn, are generated by the fast output transition speeds, making the problem worse.

Bipolar TTL makes demands on the power supply (+5V, $\pm 5\%$, with relatively high quiescent power dissipation).

⁸⁸ It’s usually less; for example, the 74HC74 *D*-type flip-flop specifies a minimum removal time (preset or clear to clock) of 5 ns, whereas the minimum data setup time is 20 ns.

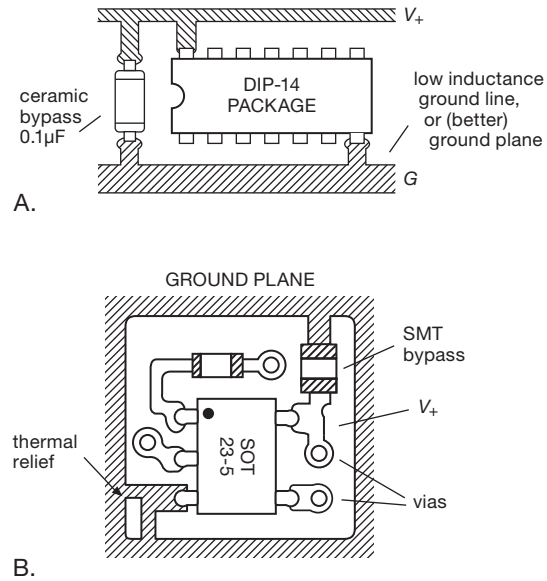


Figure 10.98. It’s always a good idea to use robust low-inductance ground wiring, with liberal use of bypass capacitors. A. For an inexpensive two-sided board you should use gridded power and ground traces. B. Better yet, use a ground plane and surface-mount ceramic bypass caps. (The top layer is usually a signal layer, with power and ground planes stacked underneath, in a multilayer PCB. We’ve shown ground on top for clarity, though you might well do this in a two-layer PCB.)

Power-supply-current spikes generated by the active pullup output circuitry generally require liberal use of power-supply bypassing, ideally one 0.1µF capacitor per chip (Figure 10.98).

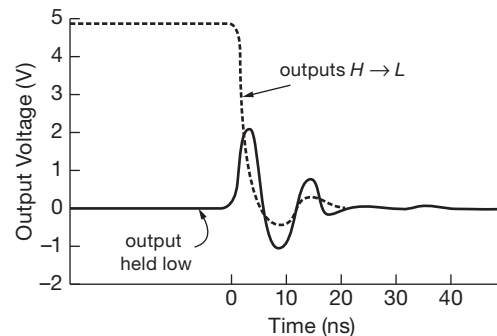


Figure 10.99. Ground bounce: 74AC244 octal buffer, driving seven 50 pF loads *H*→*L*, holding eighth output LOW. “Ground” is a copper foil plane (1 oz/ft²). (After Figure 1.1-4, *TI Advanced CMOS Logic Designer’s Handbook*.)

CMOS

CMOS inputs have traditionally been prone to damage from static electricity, with mortality rates really climbing in winter. Recent families with polysilicon gates and effective input protection networks are much more rugged than their metal-gate ancestors. CMOS inputs show a large spread in logic threshold, which can lead to problems of clock skew (§10.8.2C). Logic outputs can even exhibit double transitions when driven with slowly rising inputs. CMOS requires all unused inputs, even those of unused gate sections, to be connected to HIGH or LOW.

An interesting congenital problem with fast CMOS families is the presence of *ground bounce*: a fast CMOS chip driving its capacitive load generates enormous transient ground currents, causing the chip's ground line to jump momentarily, and thereby carrying with it LOW outputs that happen to be innocent bystanders on the same chip. Figure 10.99 shows the sort of thing you see. Notice particularly the magnitude of the effect: 1 to 2 volts is not uncommon! When you consider that a 3 ns, 5 V transition into 50 pF amounts to a transient current $I = C dV/dt = 83$ mA, and that an octal buffer might drive eight such loads simultaneously (total current of 2/3 A!), this behavior isn't surprising. When fast AC(T) logic first appeared, clothed in the traditional DIP corner-pin power and ground packaging, this problem turned out to be harder to solve than anticipated, leading to a new set of AC(T) circuits with "center-pin" power and grounds (for lower inductance). Additionally, logic IC manufacturers made design improvements to limit peak slew rates (sometimes called *edge rates*), and thus the consequent $C dV/dt$ capacitive-load currents: logic families such as AC(T)Q ("Q" for "quiet") helped considerably, with little speed compromise.

A better solution has evolved, namely the move to smaller surface-mount packages (with less lead inductance), and the widespread use of multilayer circuit boards (with dedicated power and ground layers), combined with surface-mount bypass capacitors. Recent logic ICs sometimes specify self-induced noise levels.⁸⁹ And VLSI chips with lots of pins usually dedicate multiple pins (sometimes dozens!⁹⁰) to ground. Ground-bounce problems have not been banished, though. Users should be aware of this serious problem and take measures to keep ground induc-

tance as low as possible (Figure 10.98). It's best to use circuit boards with dedicated power and ground planes, and plenty of low-inductance bypass capacitors. Better still, if you don't need the speed, stay away from the fastest logic families entirely.

B. Bizarre behavior

Bipolar logic

TTL doesn't do many really weird things.⁹¹ However, some TTL monostables will trigger on a glitch on the supply (or ground) line, and they generally behave somewhat fidgety. A circuit that works well with LS TTL may malfunction when replaced with AS TTL, because of faster edge times and consequently larger ground-line currents and ringing (74F TTL seems better in this regard). Most weird TTL operation can be traced to noise problems.

ECL involves very fast transition times, and interconnections longer than a few centimeters must be treated as terminated transmission lines (see Appendix H, and §1x.1).

CMOS

CMOS logic can drive you crazy! Open inputs on CMOS chips are bad news: you might have a circuit that intermittently misbehaves. You put your 'scope probe onto a point in the circuit, and it shows zero volts, as it should. Then the circuit works fine for a few minutes – before malfunctioning again! What happened was that the 'scope discharged the capacitance on the open input, and it took a long time to charge back up to the logic threshold.

Another amusing stunt is when a CMOS chip goes into "SCR latchup," caused by forcing an input (or output) beyond the supply rails momentarily. The resultant current (50 mA or so) through the input-protection diodes turns on a pair of parasitic cross-connected transistors that are a side effect of the "junction-isolated" CMOS process. This effectively shorts V_{DD} to ground; the chip gets hot, and you have to turn off the power supply before it will behave itself again. If you let this happen for more than a few seconds, you'll have to replace the chip. Some of the newer CMOS designs claim immunity to latchup, even with input swings 5 V *beyond* the rails, and to operate properly for input swings 1.5 volts beyond the rails.⁹²

⁹¹ A cynic might add that it doesn't do many *good* things, either.

⁹² SCR latchup happens when input *current* exceeds some threshold: manufacturers often guarantee no latchup if I_{in} is kept below some maximum "input clamp current," e.g., 20 mA. The datasheet will say something like "the input and output voltage ratings may be exceeded if the input and output current ratings are observed." This is good to know, because you can use a series input resistor to prevent latchup even with input-voltage overdrive.

⁸⁹ Sometimes called *quiet-output* logic levels, or *ringback*. These specify the maximum dynamic swing on an output that should remain in one logic state, while all other outputs from the same chip make a logic transition.

⁹⁰ Or even *hundreds*: we counted 423 ground pins on Xilinx's Virtex-7 FPGA in a 1761-pin package.

CMOS has some strange and subtle failure modes. One of the output FETs can “fail open,” giving pattern-sensitive failures that are difficult to detect. An input may begin to sink or source current. Or the whole chip may start drawing substantial supply current. Putting a $1\ \Omega$ resistor in series with each chip’s V_{DD} lead (with downstream bypassing) makes it easy to locate faulty CMOS chips that are drawing quiescent supply current (for power drivers or chips driving many outputs, use $0.1\ \Omega$ sense resistors). Most of the time you don’t bother with this precaution; but it can be a good idea if you’re making a battery-operated device, where microamps matter.

Besides the input threshold variation between chips, a single chip can exhibit different thresholds for several internal chip functions driven from a single input. For example, the RESET input of a CD4013 can bring Q' HIGH before it brings Q LOW. This means that you should not terminate a reset pulse based on the output at Q' , because the runt pulse that will be generated may actually fail to clear the flip-flop.

Open inputs would be bad news anyway, even on unused gate sections. That’s because the input can float up toward mid-supply, putting both n -channel and p -channel MOSFETs into conduction. This “class-A current” causes unwanted quiescent current (hey, CMOS is zero power, right?), and it can cause oscillations and (in some cases) enough power dissipation to destroy the IC. You can see how this goes in Figure 10.100, where we measured separately the sinking and sourcing currents in one section of a 74LVC04 inverter running on $+3.3\text{ V}$. As long as the input is within about 0.7 V of either rail, the corresponding MOSFET is fully OFF; but in between there is some simultaneous conduction, or “shoot-through” current. In this case that current peaks at about 20 mA for $V_{in} \sim 1.4\text{ V}$, causing 28 mW of dissipation in the inverter. If all six inverter sections floated, you could have nearly 200 mW of dissipation; and if the chip were running from a 5 V logic supply, the dissipation could reach destructive levels.

The maximum shoot-through current depends strongly on supply voltage, as shown in the measured data of Figure 10.101. At very low supply voltages, there is no input-voltage level that brings both MOSFETs into conduction simultaneously.

Here’s the craziest of them all: you forgot to wire up the V_{DD} pin on a CMOS chip, but the circuit works just fine! That’s because it is being powered by one of its logic inputs (via the chip’s internal input-protection diodes from input to V_{DD}). You might get away with this for a long time, but suddenly the circuit reaches a state where all the logic inputs to the chip are simultaneously LOW; the chip loses its

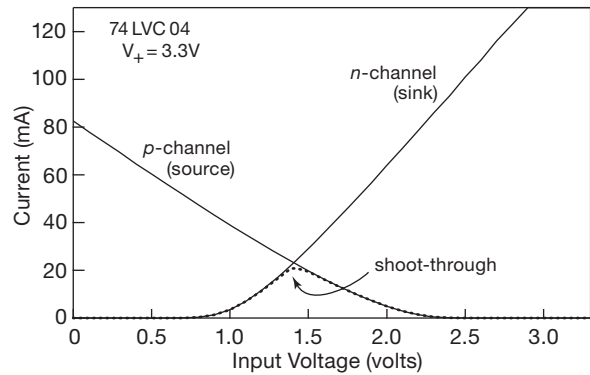


Figure 10.100. Measured sinking and sourcing currents in one section of a 'LVC04 inverter, as a function of logic input voltage. “Shoot-through current” is the colorful name given to simultaneous conduction, caused by a digital input voltage that is not near one of the rails.

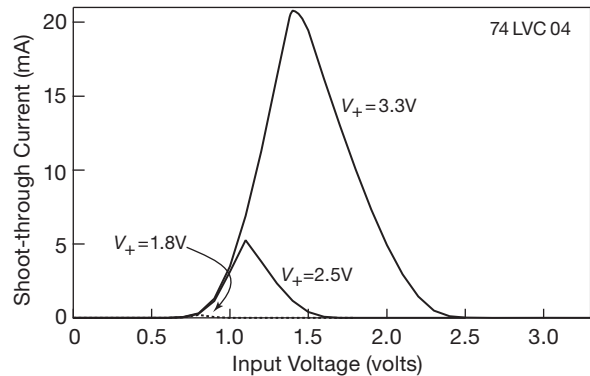


Figure 10.101. Measured shoot-through current versus digital input voltage level, for several supply voltages.

power and forgets its state. Of course, this is a bad situation anyway, since the output stage isn’t adequately powered and can’t source much current. The trouble is that this situation may produce symptoms only occasionally, and it can have you running around in circles until you figure out what’s going on.

Additional Exercises for Chapter 10

Exercise 10.31. Show how to make a JK flip-flop by using a type- D flip-flop and a 4-input MUX. *Hint:* use the address inputs for J and K .

Exercise 10.32. Design a circuit that reads out, on 7-segment digits, how many milliseconds you’ve held a button down. The device should be smart enough to reset itself each time. Use a 1.0 MHz oscillator.

Exercise 10.33. Design a reaction timer. “A” pushes his button; an LED goes on, and a counter begins counting. When “B” pushes her button, the light goes out and an LED display reads the time, in milliseconds. Be sure to design the circuit so that it will function properly even if A’s button is still held down when B’s button is pushed.

Exercise 10.34. Design a period counter: a device to measure the number of microseconds in one period of an input waveform. Use a Schmitt comparator to generate logic levels; and use a 10 MHz clock frequency. Make it work so that pushing a button initiates the next measurement.

Exercise 10.35. Add latches to the period counter, if you haven’t already.

Exercise 10.36. Now make it measure the time for 10 periods. Also, have it light an LED while it’s counting.

Exercise 10.37. Design a true electronic stopwatch. Button A starts and stops the count. Button B resets the count. The output should be of the form *xx.x* (seconds and tenths); assume that you have a 1.0 MHz square wave.

Exercise 10.38. Some stopwatches use a single button (start, stop, reset, start, etc., each time it is pushed). Design an electronic equivalent.

Exercise 10.39. Design a nice frequency counter to measure the number of cycles per second of an input waveform. Include lots of digits, latched count while counting the next interval, and a choice of 1 s, 0.1 s, or 0.01 s counting interval. You might add a good input circuit with several sensitivities, a Schmitt trigger with adjustable hysteresis and trigger point (use a fast comparator), and a logic signal input for TTL signals. How about a BCD output? Multiplex the digits on output, as well as parallel output. Spend some time on this one.⁹³

Exercise 10.40. Design a circuit, using LVC logic at 3.3 V, to time a speeding bullet. The projectile breaks a thin wire stretched across its path; then, some measured distance farther along its path, it breaks a second wire. Beware of problems like “contact bounce.” Assume that you have a 10 MHz logic square wave, and design your circuit to read out, in microseconds (four digits), the time interval between breaking the two wires. A pushbutton should reset the circuit for the next shot.

Exercise 10.41. Invent a circuit to keep a running sum of successive 4-bit binary numbers that are input to it. Keep your result to only 4 bits (i.e., perform a sum modulo-16).⁹⁴

Now add another feature to the circuit, namely, an output bit that is 1 if the *total number* of 1 bits in all the input numbers (since the last reset input) is odd, 0 if even. *Hint:* an XOR “parity tree” will tell you if the sum of 1s in each number is odd; figure it out from there.

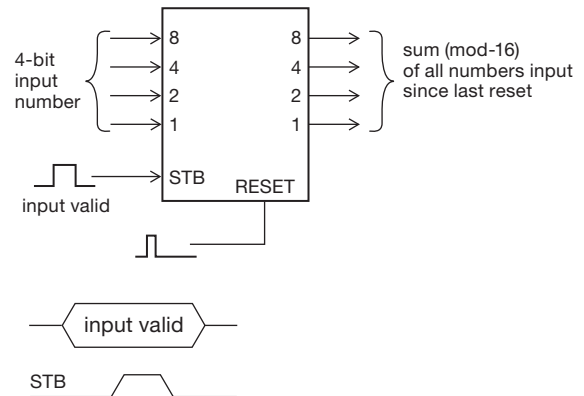


Figure 10.102. Checksum circuit block diagram.

Exercise 10.42. In Exercise 10.15 you designed a 2×2 multiplier by using a Karnaugh map for each output bit. In this problem you are to accomplish the same task by the process of “shift and add.” Begin by writing out the product the way you would in elementary school (Figure 10.103). This process has a simple repeating pattern, requiring 2-input gates (what kind?) to generate the intermediate terms a_0b_0 , etc., and 1-bit “half-adders” (adders that have a carry out but no carry in) to sum the intermediate terms.

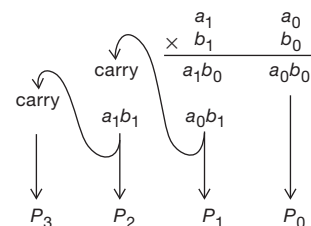


Figure 10.103. How to multiply.

⁹³ H & H did – this was one of our first joint projects, some forty years ago.

⁹⁴ More sophisticated variants of such a sum, e.g., a CRC code (cyclic-redundancy checksum) are useful as a validity check against errors introduced into data files intended for storage or transmission.

Review of Chapter 10

An A-to-H summary of what we have learned in Chapter 10. This summary reviews basic principles and facts in Chapter 10, but it does not cover application circuit diagrams and practical engineering advice presented there.

¶A. Digital Voltage Levels.

In contrast to analog electronics, where valid signals occupy some range of voltages (e.g., an op-amp's $\pm 10\text{ V}$ output swing), the signals in *digital* electronics are confined to two (usually) states, called HIGH and LOW (or 1 and 0, or TRUE and FALSE). Each state has a nominal voltage level, and a defined valid range over which it will be properly interpreted by the input of a driven digital device (§10.1.2B, also §12.1.2). The signal may be single-ended (e.g., a $+3.3\text{ V}$ CMOS output, with nominal levels of 0 and $+3.3\text{ V}$, see Figure 10.2), or it may be a differential pair (e.g., LVDS, whose current-mode drive produces a nominal 300 mV swing centered around $+1.25\text{ V}$, seen later in Figure 12.135). There are numerous *families* of digital logic, distinguished by their supply voltage range, input thresholds, speed, power dissipation, and the like; see ¶G.

¶B. Meaning of Digital Bits.

Individual logic bits may represent part of a larger quantity (e.g., bits in a data byte), or they may represent a state (e.g., RESET, or RUN). A group of n bits can be sent simultaneously as a *parallel* quantity (on n wires) or sequentially in time (on a single wire or pair) as a *serial* string, comprising a parallel or serial *bus*. Numeric quantities are represented as unsigned integers, 2's-complement integers, or floating-point numbers (§10.1.3), whereas alphanumeric characters use single-byte ASCII coding (or its two-byte extensions). Base-16 hexadecimal notation (0–9, a–f) is ordinarily used to write multibit binary quantities, with each hex character representing four binary bits (thus two hex characters per byte). Whether representing multibit quantities or single state bits, digital voltage levels in a circuit can be assigned as active-HIGH or active-LOW. For example, one might have an active-LOW signal to reset a microprocessor; it would be called RESET', and would ordinarily stay *disasserted* at the HIGH voltage state, going LOW only when it asserts a reset operation.

¶C. Combinational Logic: Gates.

Digital circuits whose state depends only on present inputs (i.e., not on past history) are called *combinational*. Their logical operations are performed with *gates*, whose basic forms (§10.1.4) are AND, OR, and NOT (or *invert*),

and whose standard graphic symbols and truth tables are shown in Figures 10.4, 10.5, and 10.6. These can be combined to form functions such as NAND and NOR (inverted-output AND and OR, Figure 10.8), or exclusive-OR (XOR, Figure 10.9). And they can be strung together to make any specified logical combinational output (or outputs) from a set of inputs.

¶D. Sequential Logic: Flip-flops.

The basic element of a sequential logic circuit is the *flip-flop* (§10.4), a device that maintains its state in the absence of external inputs; it is a “1-bit memory.” The state of a circuit with flip-flops depends on both the present inputs and the previous state. For the ubiquitous *type-D* flip-flop (“D-flop”), the logic level present at the D (data) input at the time of⁹⁵ a clocking transition (edge) is captured and presented to the Q output. Once you have flip-flops, the world is your oyster – you can make counters, registers, arbitrary “finite-state machines” – and (drumroll) *computers!*

When a flip-flop is augmented with an RC , you get a *monostable*⁹⁶ *multivibrator* (or “one-shot”) timing circuit, which generates an output logic pulse (of width set by the time constant RC) when triggered by an input edge. The one-shot is an example of a mixed-signal circuit, i.e., a combination of analog and digital techniques.

¶E. Sequential Logic: Registers and Counters.

A clocked D -flop captures and holds one bit; a collection of n D -flops with a common clock input is called a *register* (§10.5.1); it captures and holds n bits (a byte, for $n=8$). When combined with memory and gates, it can form a *finite state machine* (FSM, see Figure 10.64), just a step away from the microprocessor.

If the output bits Q_i of a D -register drive the subsequent inputs (i.e., Q_i is connected to D_{i+1}), you've got a *shift register*; these are handy for parallelizing a serial stream, or vice versa (if loaded in parallel and then shifted out the end). An interesting shift-register application is the generation of pseudo-random (noise-like) bit sequences, see §§8.12.4A and 13.14.

A collection of n D -flops can instead be connected to make an n -bit *counter*, such that successive input transitions cause the n -bit number represented by Q_0 – Q_{n-1} to increment.⁹⁷ Simplest is the *ripple counter* (§10.4.2E), in

⁹⁵ More precisely, present and stable from a *setup time* t_s before the edge to a *hold time* t_h after the edge, to ensure proper operation.

⁹⁶ One-side stable, as contrasted with the ordinary *bistable* flip-flop (stable in either state), or the *astable* oscillator (stable in neither state).

⁹⁷ Or to decrement, for a *down counter*.

which the Q_i outputs drive the successive clock inputs; the alternative is a *synchronous* counter, in which all clock inputs are driven by the input signal, and gate logic is rigged up to present to the D inputs the levels corresponding to the next state. The latter is just a special case of a synchronous FSM. Synchronous counters (and synchronous systems in general) are favored, having the pleasant properties of simultaneous state changes (apart from *skew*), and of lack of digital noise in the setup interval before each system clock.

¶F. Standard Logic and Programmable Logic.

Digital logic is available as standard pre-wired functions (gates, flip-flops, counters, registers), in packages going from popular single-gate mini-logic “glue” in 5-pin or 6-pin packages, through the long-popular mid-sized (14-, 16-, or 20-pin) packages, to large multibyte-wide bus drivers in 48-pin or 96-pin packages; see, for example, the listings in Table 10.3 on page 716 (gates) and Table 10.5 on page 742 (counters).

These are good enough for many tasks; but an attractive alternative, particularly in a complex system, is the use of *programmable logic devices* (PLDs). These contain large numbers of uncommitted gates and flip-flops, whose interconnections are user-configurable (and re-configurable); see §10.5.4. The category known as *cPLDs* (complex PLDs) contain from a few dozen to a few hundred “macrocells” (a flip-flop and a collection of gates) and a few thousand gates, along with the non-volatile program memory, and they can replace much of the “random logic” (the tangle of gates and flip-flops) in a digital system; they are also ideal for implementing state machines. At a higher level of integration, the *field-programmable gate array* (FPGA) reigns supreme; the largest of these incorporate billions of transistors implementing millions of flip-flops, along with memory, serial transceivers, and all the resources needed to implement a microprocessor.

Most designers use a hardware description language (HDL), running on a PC platform, to specify the PLD’s function, but there are available schematic-entry tools as well; these toolsets allow you to simulate the programmed functions, to verify that you’ve programmed what you intended. Contemporary PLDs are programmed in-circuit, usually via a JTAG serial port. We discuss PLDs in greater detail in Chapter 11.

Any discussion of digital logic must include *microcontrollers*, the increasingly capable and inexpensive computers-on-a-chip that can replace discrete and programmable logic devices in digital devices. Think of em-

bedded microcontrollers not as exotic creatures, but simply as circuit components – they cost less than a precision op-amp, and they can do wonders. They are the subject of Chapter 15.

¶G. Logic Families.

Digital logic functions, whether standard logic or programmable logic, can be implemented in ICs with a variety of on-chip transistor technologies (§10.2). Earlier legacy families of standard logic (bipolar RTL, DTL, and TTL) have yielded their turf to the now-dominant CMOS logic families. The latter include the traditional 5 V HC[T] and AC[T] families (the T suffix indicates TTL thresholds, around 1.4 V), the high-voltage (but very slow) 4000B series, and a proliferation of CMOS families intended for lower-voltage operation; the latter include popular families like LVC (1.8–5 V) and AUC (1.2–2.5 V), among the dozens of choices. See Figures 10.22, 10.26, 10.27, and (in Chapter 12) §12.1 and Figure 12.2.

Programmable logic devices (cPLDs, FPGAs) and microcontrollers are invariably built with CMOS, with supply voltage ranging from as low as 1 V to as high as 5 V, and reflected in their speed and power dissipation. It is common to have separate “core” and I/O supply rails in many of these devices; for example a microcontroller or FPGA might have a 1.2 V core and a 1.6–3.6 V I/O (the latter for compatibility with external devices).

All these variations can be confusing. Chapter 12 deals in detail with logic family interfacing. At a simple level, though, what you need to know (summarized in §12.1.3, and encapsulated in Figure 12.9) is that (a) you can always make a direct connection between logic running at the same voltage; (b) logic running from a higher voltage (e.g., +5 V) can drive lower-voltage logic if the latter’s input is “tolerant”; and (c) lower-voltage logic can drive higher-voltage logic if the latter has “TTL thresholds” and the former is powered from at least +2.5 V.

¶H. Digital Logic Cautions.

Digital design nicely sidesteps the complications inherent in linear circuits (biasing, thermal stability, etc.), and it’s easy to become complacent, and lazy. But there are plenty of hazards in the digital playground, see §10.8. Among the *dc problems* are logical lockup states, SCR latchup, and undetermined startup states. More sinister are *switching problems* such as logic races and runt pulses, metastable states, clock and data skew, ground bounce, and power- and ground-conducted switching noise.