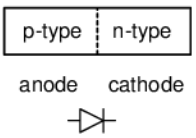
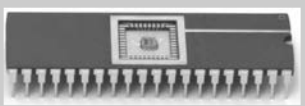


Figure 1.27 The p-n junction diode structure and symbol

Figure 1.28 Capacitor symbol

Technicians in an Intel clean room wear Gore-Tex bunny suits to prevent particulates from their hair, skin, and clothing from contaminating the microscopic transistors on silicon wafers (© 2006, Intel Corporation. Reproduced by permission).



A 40-pin dual-inline package (DIP) contains a small chip (scarcely visible) in the center that is connected to 40 metal pins, 20 on a side, by gold wires thinner than a strand of hair (photograph by Kevin Mapp. © Harvey Mudd College).

current flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is *reverse biased*, and no current flows. The diode symbol intuitively shows that current only flows in one direction.

1.7.3 Capacitors

A *capacitor* consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$. The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = Q/V$. The capacitance is proportional to the size of the conductors and inversely proportional the distance between them. The symbol for a capacitor is shown in Figure 1.28.

Capacitance is important because charging or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate. Speed and energy will be discussed throughout this book.

1.7.4 nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron[2] in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called *chips* or *dice* that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.

The MOSFET sandwich consists of a conducting layer called the *gate* on top of an insulating layer of *silicon dioxide* (SiO_2) on top of the silicon wafer, called the *substrate*. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate, because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called *oxide* in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a *dielectric* separates the metal and semiconductor plates.

[2] 1 μm = 1 micron = 10^{-6} m.

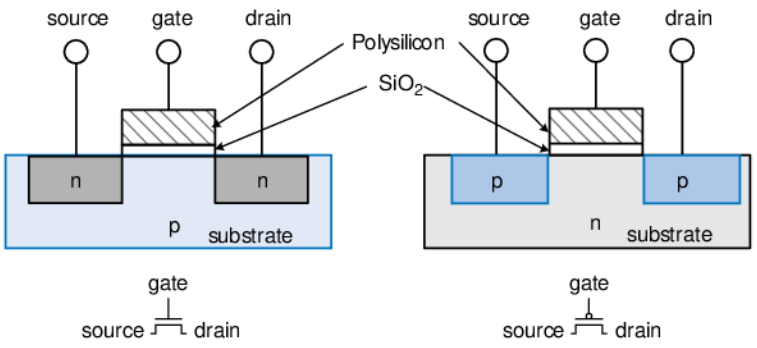


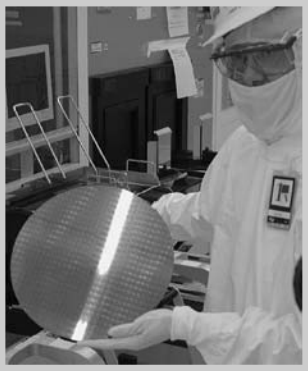
Figure 1.29 nMOS and pMOS transistors

The source and drain terminals are physically symmetric. However, we say that charge flows from the source to the drain. In an nMOS transistor, the charge is carried by electrons, which flow from negative voltage to positive voltage. In a pMOS transistor, the charge is carried by holes, which flow from positive voltage to negative voltage. If we draw schematics with the most positive voltage at the top and the most negative at the bottom, the source of (negative) charges in an nMOS transistor is the bottom terminal and the source of (positive) charges in a pMOS transistor is the top terminal.

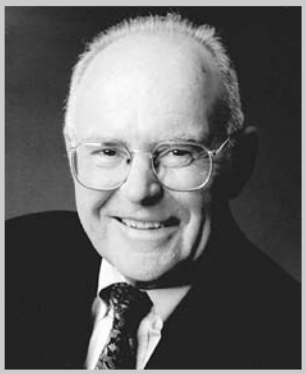
There are two flavors of MOSFETs: nMOS and pMOS (pronounced "n-moss" and "p-moss"). Figure 1.29 shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent to the gate called the *source* and the *drain* and are built on a p-type semiconductor substrate. The *pMOS* transistors are just the opposite, consisting of p-type source and drain regions in an n-type *substrate*.

A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, the lowest voltage in the system. First, consider the situation when the gate is also at 0 V, as shown in Figure 1.30(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in Figure 1.30(b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the *channel*. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the *threshold voltage*, V_t, and is typically 0.3 to 0.7 V.



A technician holds a 12-inch wafer containing hundreds of microprocessor chips (© 2006, Intel Corporation. Reproduced by permission).



Gordon Moore, 1929–. Born in San Francisco. Received a B.S. in chemistry from UC Berkeley and a Ph.D. in chemistry and physics from Caltech. Cofounded Intel in 1968 with Robert Noyce. Observed in 1965 that the number of transistors on a computer chip doubles every year. This trend has become known as *Moore's Law*. Since 1975, transistor counts have doubled every two years.

A corollary of Moore's Law is that microprocessor performance doubles every 18 to 24 months. Semiconductor sales have also increased exponentially. Unfortunately, power consumption has increased exponentially as well (© 2006, Intel Corporation. Reproduced by permission).

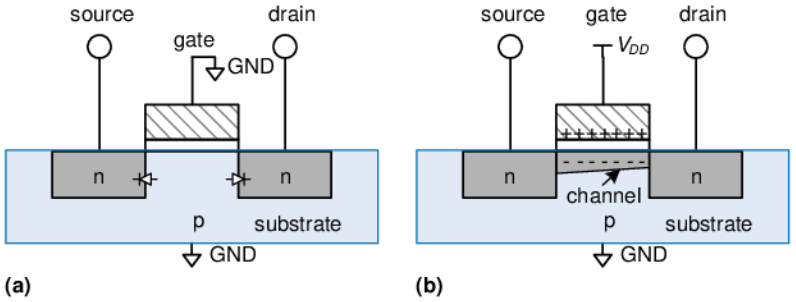


Figure 1.30 nMOS transistor operation

pMOS transistors work in just the opposite fashion, as might be guessed from the bubble on their symbol. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the drain will only swing between 0 and $V_{DD} - V_t$. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called *wells* where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or *CMOS*. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS processes give us two types of electrically controlled switches, as shown in Figure 1.31. The voltage at the gate (g) regulates the flow of current between the source (s) and drain (d). nMOS transistors are OFF when the gate is 0 and ON when the gate is 1.

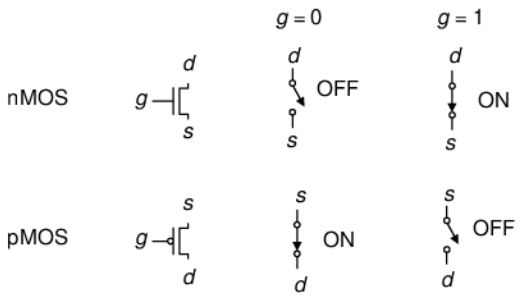


Figure 1.31 Switch models of MOSFETs

pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

1.7.5 CMOS NOT Gate

Figure 1.32 shows a schematic of a NOT gate built with CMOS transistors. The triangle indicates GND, and the flat bar indicates V_{DD}; these labels will be omitted from future schematics. The nMOS transistor, *N1*, is connected between GND and the *Y* output. The pMOS transistor, *P1*, is connected between V_{DD} and the *Y* output. Both transistor gates are controlled by the input, *A*.

If $A = 0$, *N1* is OFF and *P1* is ON. Hence, *Y* is connected to V_{DD} but not to GND, and is pulled up to a logic 1. *P1* passes a good 1. If $A = 1$, *N1* is ON and *P1* is OFF, and *Y* is pulled down to a logic 0. *N1* passes a good 0. Checking against the truth table in Figure 1.12, we see that the circuit is indeed a NOT gate.

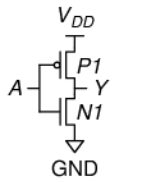


Figure 1.32 NOT gate schematic

1.7.6 Other CMOS Logic Gates

Figure 1.33 shows a schematic of a two-input NAND gate. In schematic diagrams, wires are always joined at three-way junctions. They are joined at four-way junctions only if a dot is shown. The nMOS transistors *N1* and *N2* are connected in series; both nMOS transistors must be ON to pull the output down to GND. The pMOS transistors *P1* and *P2* are in parallel; only one pMOS transistor must be ON to pull the output up to V_{DD}. Table 1.6 lists the operation of the pull-down and pull-up networks and the state of the output, demonstrating that the gate does function as a NAND. For example, when $A = 1$ and $B = 0$, *N1* is ON, but *N2* is OFF, blocking the path from *Y* to GND. *P1* is OFF, but *P2* is ON, creating a path from V_{DD} to *Y*. Therefore, *Y* is pulled up to 1.

Figure 1.34 shows the general form used to construct any inverting logic gate, such as NOT, NAND, or NOR. nMOS transistors are good at passing 0's, so a pull-down network of nMOS transistors is placed between the output and GND to pull the output down to 0. pMOS transistors are

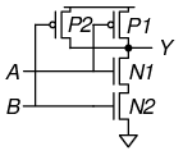


Figure 1.33 Two-input NAND gate schematic

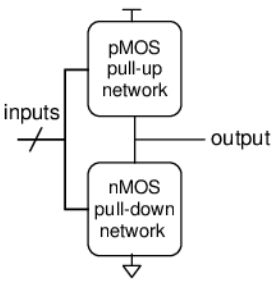


Figure 1.34 General form of an inverting logic gate

Table 1.6 NAND gate operation

A	*B*	Pull-Down Network	Pull-Up Network	*Y*
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

Experienced designers claim that electronic devices operate because they contain *magic smoke*. They confirm this theory with the observation that if the magic smoke is ever let out of the device, it ceases to work.

good at passing 1's, so a pull-up network of pMOS transistors is placed between the output and V_{DD} to pull the output up to 1. The networks may consist of transistors in series or in parallel. When transistors are in parallel, the network is ON if either transistor is ON. When transistors are in series, the network is ON only if both transistors are ON. The slash across the input wire indicates that the gate may receive multiple inputs.

If both the pull-up and pull-down networks were ON simultaneously, a *short circuit* would exist between V_{DD} and GND. The output of the gate might be in the forbidden zone and the transistors would consume large amounts of power, possibly enough to burn out. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output *floats*. Its value is again undefined. Floating outputs are usually undesirable, but in Section 2.6 we will see how they can occasionally be used to the designer's advantage.

In a properly functioning logic gate, one of the networks should be ON and the other OFF at any given time, so that the output is pulled HIGH or LOW but not shorted or floating. We can guarantee this by using the rule of *conduction complements*. When nMOS transistors are in series, the pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

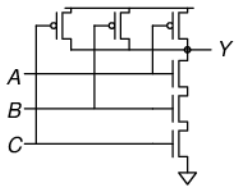


Figure 1.35 Three-input NAND gate schematic

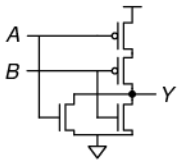


Figure 1.36 Two-input NOR gate schematic

Example 1.20 THREE-INPUT NAND SCHEMATIC

Draw a schematic for a three-input NAND gate using CMOS transistors.

Solution: The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel. Such a gate is shown in Figure 1.35; you can verify the function by checking that it has the correct truth table.

Example 1.21 TWO-INPUT NOR SCHEMATIC

Draw a schematic for a two-input NOR gate using CMOS transistors.

Solution: The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series. Such a gate is shown in Figure 1.36.

Example 1.22 TWO-INPUT AND SCHEMATIC

Draw a schematic for a two-input AND gate.

Solution: It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT, as shown in Figure 1.37.

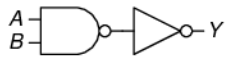


Figure 1.37 Two-input AND gate schematic

1.7.7 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well. Recall that nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well. Figure 1.38 shows such a circuit, called a *transmission gate* or *pass gate*. The two sides of the switch are called A and B because a switch is bidirectional and has no preferred input or output side. The control signals are called *enable*s, EN and $\overline{EN}$. When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so A and B are not connected. When $EN = 1$ and $\overline{EN} = 0$, the transmission gate is ON or enabled, and any logic value can flow between A and B.

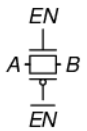


Figure 1.38 Transmission gate

1.7.8 Pseudo-nMOS Logic

An N-input CMOS NOR gate uses N nMOS transistors in parallel and N pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel. Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore the parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.39. This pMOS transistor is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output, Y, HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls Y down close enough to GND to produce a logic 0.

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs. For example, Figure 1.40 shows a pseudo-nMOS four-input NOR. Pseudo-nMOS gates are useful for certain memory and logic arrays discussed in Chapter 5. The disadvantage is that a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

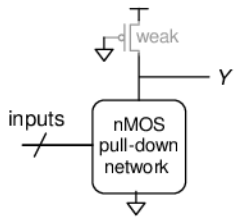


Figure 1.39 Generic pseudo-nMOS gate

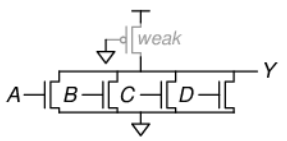


Figure 1.40 Pseudo-nMOS four-input NOR gate

Pseudo-nMOS gates got their name from the 1970's, when manufacturing processes only had nMOS transistors. A weak nMOS transistor was used to pull the output HIGH because pMOS transistors were not available.

1.8 POWER CONSUMPTION*

Power consumption is the amount of energy used per unit time. Power consumption is of great importance in digital systems. The battery life of portable systems such as cell phones and laptop computers is limited by power consumption. Power is also significant for systems that are plugged in, because electricity costs money and because the system will overheat if it draws too much power.

Digital systems draw both *dynamic* and *static* power. Dynamic power is the power used to charge capacitance as signals change between 0 and 1. Static power is the power used even when signals do not change and the system is idle.

Logic gates and the wires that connect them have capacitance. The energy drawn from the power supply to charge a capacitance C to voltage V_{DD} is CV_{DD}^2. If the voltage on the capacitor switches at frequency f (i.e., f times per second), it charges the capacitor $f/2$ times and discharges it $f/2$ times per second. Discharging does not draw energy from the power supply, so the dynamic power consumption is

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^2 f \tag{1.4}$$

Electrical systems draw some current even when they are idle. When transistors are OFF, they leak a small amount of current. Some circuits, such as the pseudo-nMOS gate discussed in Section 1.7.8, have a path from V_{DD} to GND through which current flows continuously. The total static current, I_{DD}, is also called the *leakage current* or the *quiescent supply current* flowing between V_{DD} and GND. The static power consumption is proportional to this static current:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1.5}$$

Example 1.23 POWER CONSUMPTION

A particular cell phone has a 6 watt-hour (W-hr) battery and operates at 1.2 V. Suppose that, when it is in use, the cell phone operates at 300 MHz and the average amount of capacitance in the chip switching at any given time is 10 nF (10^{-8} Farads). When in use, it also broadcasts 3 W of power out of its antenna. When the phone is not in use, the dynamic power drops to almost zero because the signal processing is turned off. But the phone also draws 40 mA of quiescent current whether it is in use or not. Determine the battery life of the phone (a) if it is not being used, and (b) if it is being used continuously.

Solution: The static power is $P_{static} = (0.040\ A)(1.2\ V) = 48\ mW$. If the phone is not being used, this is the only power consumption, so the battery life is (6 W-hr)/(0.048 W) = 125 hours (about 5 days). If the phone is being used, the dynamic power is $P_{dynamic} = (0.5)(10^{-8}\ F)(1.2\ V)^2(3 \times 10^8\ Hz) = 2.16\ W$. Together with the static and broadcast power, the total active power is 2.16 W + 0.048 W + 3 W = 5.2 W, so the battery life is 6 W-hr/5.2 W = 1.15 hours. This example somewhat oversimplifies the actual operation of a cell phone, but it illustrates the key ideas of power consumption.

1.9 SUMMARY AND A LOOK AHEAD

There are 10 kinds of people in this world: those who can count in binary and those who can't.

This chapter has introduced principles for understanding and designing complex systems. Although the real world is analog, digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH. Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

- **NOT:** TRUE when input is FALSE
- **AND:** TRUE when all inputs are TRUE
- **OR:** TRUE when any inputs are TRUE
- **XOR:** TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches. nMOS transistors turn ON when the gate is 1. pMOS transistors turn ON when the gate is 0.

In Chapters 2 through 5, we continue the study of digital logic. Chapter 2 addresses *combinational logic*, in which the outputs depend only on the current inputs. The logic gates introduced already are examples of combinational logic. You will learn to design circuits involving multiple gates to implement a relationship between inputs and outputs specified by a truth table or Boolean equation. Chapter 3 addresses *sequential logic*, in which the outputs depend on both current and past inputs. *Registers* are common sequential elements that remember their previous input. *Finite state machines*, built from registers and combinational logic, are a powerful way to build complicated systems in a systematic fashion. We also study timing of digital systems to analyze how fast the systems can operate. Chapter 4 describes hardware description languages (HDLs). HDLs are related to conventional programming languages but are used to simulate and build hardware rather than software. Most digital systems today are designed with HDLs. Verilog

and VHDL are the two prevalent languages, and they are covered side-by-side in this book. Chapter 5 studies other combinational and sequential building blocks such as adders, multipliers, and memories.

Chapter 6 shifts to computer architecture. It describes the MIPS processor, an industry-standard microprocessor used in consumer electronics, some Silicon Graphics workstations, and many communications systems such as televisions, networking hardware and wireless links. The MIPS architecture is defined by its registers and assembly language instruction set. You will learn to write programs in assembly language for the MIPS processor so that you can communicate with the processor in its native language.

Chapters 7 and 8 bridge the gap between digital logic and computer architecture. Chapter 7 investigates microarchitecture, the arrangement of digital building blocks, such as adders and registers, needed to construct a processor. In that chapter, you learn to build your own MIPS processor. Indeed, you learn three microarchitectures illustrating different trade-offs of performance and cost. Processor performance has increased exponentially, requiring ever more sophisticated memory systems to feed the insatiable demand for data. Chapter 8 delves into memory system architecture and also describes how computers communicate with peripheral devices such as keyboards and printers.

Exercises

Exercise 1.1 Explain in one paragraph at least three levels of abstraction that are used by

a) biologists studying the operation of cells.

b) chemists studying the composition of matter.

Exercise 1.2 Explain in one paragraph how the techniques of hierarchy, modularity, and regularity may be used by

a) automobile designers.

b) businesses to manage their operations.

Exercise 1.3 Ben Bitdiddle is building a house. Explain how he can use the principles of hierarchy, modularity, and regularity to save time and money during construction.

Exercise 1.4 An analog voltage is in the range of 0–5 V. If it can be measured with an accuracy of ± 50 mV, at most how many bits of information does it convey?

Exercise 1.5 A classroom has an old clock on the wall whose minute hand broke off.

a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time?

b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

Exercise 1.6 The Babylonians developed the *sexagesimal* (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the number 4000_{10} in sexagesimal?

Exercise 1.7 How many different numbers can be represented with 16 bits?

Exercise 1.8 What is the largest unsigned 32-bit binary number?

Exercise 1.9 What is the largest 16-bit binary number that can be represented with

a) unsigned numbers?

b) two's complement numbers?

c) sign/magnitude numbers?

Exercise 1.10 What is the smallest (most negative) 16-bit binary number that can be represented with

a) unsigned numbers?

b) two's complement numbers?

c) sign/magnitude numbers?

Exercise 1.11 Convert the following unsigned binary numbers to decimal.

a) 1010_2

b) 110110_2

c) 11110000_2

d) 0001100010100111_2

Exercise 1.12 Repeat Exercise 1.11, but convert to hexadecimal.

Exercise 1.13 Convert the following hexadecimal numbers to decimal.

a) $A5_{16}$

b) $3B_{16}$

c) $FFFF_{16}$

d) $D0000000_{16}$

Exercise 1.14 Repeat Exercise 1.13, but convert to unsigned binary.

Exercise 1.15 Convert the following two's complement binary numbers to decimal.

a) 1010_2

b) 110110_2

c) 01110000_2

d) 10011111_2

Exercise 1.16 Repeat Exercise 1.15, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.17 Convert the following decimal numbers to unsigned binary numbers.

a) 42_{10}

b) 63_{10}

c) 229_{10}

d) 845_{10}

Exercise 1.18 Repeat Exercise 1.17, but convert to hexadecimal.

Exercise 1.19 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

a) 42_{10}

b) -63_{10}

c) 124_{10}

d) -128_{10}

e) 133_{10}

Exercise 1.20 Repeat Exercise 1.19, but convert to 8-bit sign/magnitude numbers.

Exercise 1.21 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

a) 0101_2

b) 1010_2

Exercise 1.22 Repeat Exercise 1.21 if the numbers are unsigned rather than two's complement.

Exercise 1.23 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.17 to octal.

Exercise 1.24 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

a) 42_8

b) 63_8

c) 255_8

d) 3047_8

Exercise 1.25 How many 5-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.26 How many bytes are in a 32-bit word? How many nibbles are in the word?

Exercise 1.27 How many bytes are in a 64-bit word?

Exercise 1.28 A particular DSL modem operates at 768 kbits/sec. How many bytes can it receive in 1 minute?

Exercise 1.29 Hard disk manufacturers use the term "megabyte" to mean 10^6 bytes and "gigabyte" to mean 10^9 bytes. How many real GBs of music can you store on a 50 GB hard disk?

Exercise 1.30 Estimate the value of 2^{31} without using a calculator.

Exercise 1.31 A memory on the Pentium II microprocessor is organized as a rectangular array of bits with 2^8 rows and 2^9 columns. Estimate how many bits it has without using a calculator.

Exercise 1.32 Draw a number line analogous to Figure 1.11 for 3-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.33 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows a 4-bit result.

a) $1001_2 + 0100_2$

b) $1101_2 + 1011_2$

Exercise 1.34 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows an 8-bit result.

a) $10011001_2 + 01000100_2$

b) $11010010_2 + 10110110_2$

Exercise 1.35 Repeat Exercise 1.34, assuming that the binary numbers are in two's complement form.

Exercise 1.36 Convert the following decimal numbers to 6-bit two's complement binary numbers and add them. Indicate whether or not the sum overflows a 6-bit result.

a) $16_{10} + 9_{10}$

b) $27_{10} + 31_{10}$

c) $-4_{10} + 19_{10}$

d) $3_{10} + -32_{10}$

e) $-16_{10} + -9_{10}$

f) $-27_{10} + -31_{10}$

Exercise 1.37 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

a) $7_{16} + 9_{16}$

b) $13_{16} + 28_{16}$

c) $AB_{16} + 3E_{16}$

d) $8F_{16} + AD_{16}$

Exercise 1.38 Convert the following decimal numbers to 5-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 5-bit result.

a) $9_{10} - 7_{10}$

b) $12_{10} - 15_{10}$

c) $-6_{10} - 11_{10}$

d) $4_{10} - -8_{10}$

Exercise 1.39 In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111, 1 as 10000, and so forth. Biased number systems are sometimes used in floating point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}.

a) What decimal value does the binary number 10000010_2 represent?

b) What binary number represents the value 0?

c) What is the representation and value of the most negative number?

d) What is the representation and value of the most positive number?

Exercise 1.40 Draw a number line analogous to Figure 1.11 for 3-bit biased numbers with a bias of 3 (see Exercise 1.39 for a definition of biased numbers).

Exercise 1.41 In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

a) Write 289_{10} in BCD.

b) Convert 100101010001_{BCD} to decimal.

c) Convert 01101001_{BCD} to binary.

d) Explain why BCD might be a useful way to represent numbers.

Exercise 1.42 A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number system: 325 + 42 = 411. If this equation is correct, how many fingers would you expect Martians have?

Exercise 1.43 Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, "All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation." Alyssa disagrees. She says, "No, but all such numbers have an even number of 1's in their representation." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.44 Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, "I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result." Alyssa says, "No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.45 Write a program in your favorite language (e.g., C, Java, Perl) to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should print the decimal equivalent.

Exercise 1.46 Repeat Exercise 1.45 but convert from decimal to hexadecimal.

Exercise 1.47 Repeat Exercise 1.45 but convert from an arbitrary base b_1 to another base b_2, as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9. The user should enter b_1, b_2, and then the number to convert in base b_1. The program should print the equivalent number in base b_2.

Exercise 1.48 Draw the symbol, Boolean equation, and truth table for

a) a three-input OR gate.

b) a three-input exclusive OR (XOR) gate.

c) a four-input XNOR gate

Exercise 1.49 A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.41.

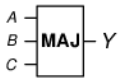


Figure 1.41 Three-input majority gate

Exercise 1.50 A three-input *AND-OR (AO) gate* shown in Figure 1.42 produces a TRUE output if both *A* and *B* are TRUE, or if *C* is TRUE. Complete a truth table for the gate.

Figure 1.42 Three-input AND-OR gate

Exercise 1.51 A three-input *OR-AND-INVERT (OAI) gate* shown in Figure 1.43 produces a FALSE input if *C* is TRUE and *A* or *B* is TRUE. Otherwise it produces a TRUE output. Complete a truth table for the gate.

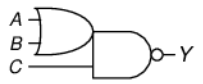


Figure 1.43 Three-input OR-AND-INVERT gate

Exercise 1.52 There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

Exercise 1.53 How many different truth tables exist for Boolean functions of *N* variables?

Exercise 1.54 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.44 would serve as an inverter? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

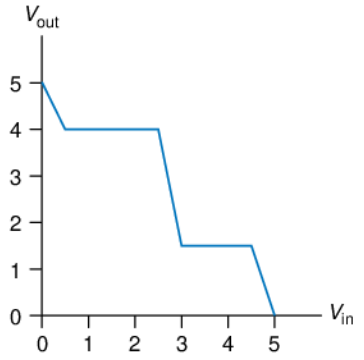


Figure 1.44 DC transfer characteristics

Exercise 1.55 Repeat Exercise 1.54 for the transfer characteristics shown in Figure 1.45.

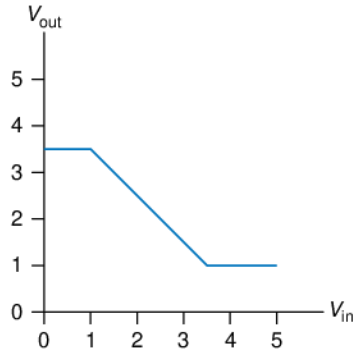


Figure 1.45 DC transfer characteristics

Exercise 1.56 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.46 would serve as a buffer? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

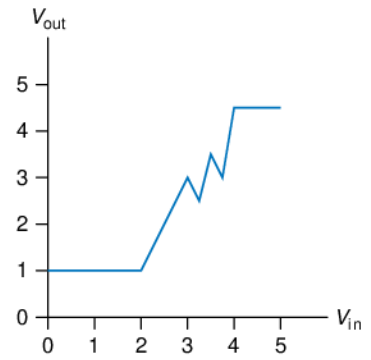


Figure 1.46 DC transfer characteristics

Exercise 1.57 Ben Bitdiddle has invented a circuit with the transfer characteristics shown in Figure 1.47 that he would like to use as a buffer. Will it work? Why or why not? He would like to advertise that it is compatible with LVCMOS and LVTTL logic. Can Ben's buffer correctly receive inputs from those logic families? Can its output properly drive those logic families? Explain.

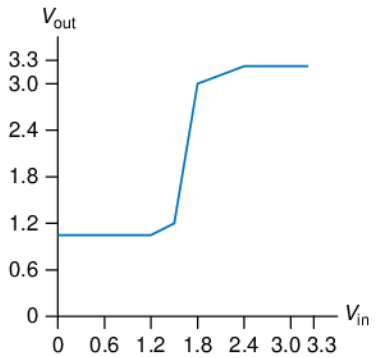


Figure 1.47 Ben's buffer DC transfer characteristics

Exercise 1.58 While walking down a dark alley, Ben Bitdiddle encounters a two-input gate with the transfer function shown in Figure 1.48. The inputs are *A* and *B* and the output is *Y*.

a) What kind of logic gate did he find?

b) What are the approximate high and low logic levels?

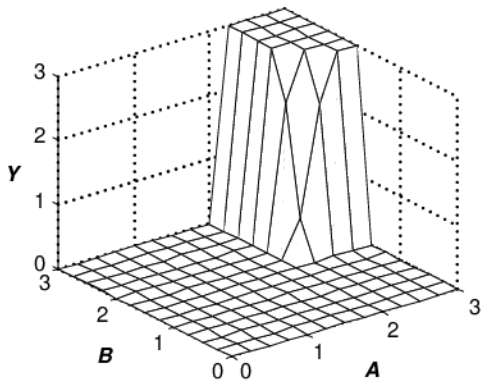


Figure 1.48 **Two-input DC transfer characteristics**

Exercise 1.59 Repeat Exercise 1.58 for Figure 1.49.

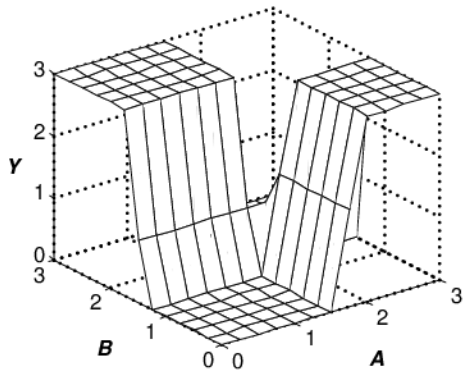


Figure 1.49 **Two-input DC transfer characteristics**

Exercise 1.60 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

a) A four-input NAND gate.

b) A three-input OR-AND-INVERT gate (see Exercise 1.51).

c) A three-input AND-OR gate (see Exercise 1.50).

Exercise 1.61 A *minority gate* produces a TRUE output if and only if fewer than half of its inputs are TRUE. Otherwise it produces a FALSE output. Sketch a transistor-level circuit for a CMOS minority gate. Use a minimum number of transistors.

Exercise 1.62 Write a truth table for the function performed by the gate in Figure 1.50. The truth table should have two inputs, A and B. What is the name of this function?

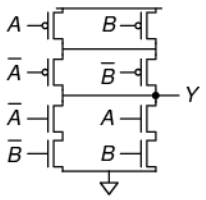


Figure 1.50 Mystery schematic

Exercise 1.63 Write a truth table for the function performed by the gate in Figure 1.51. The truth table should have three inputs, A, B, and C.

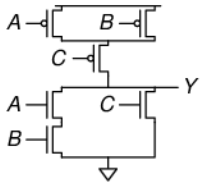


Figure 1.51 Mystery schematic

Exercise 1.64 Implement the following three-input gates using **only** pseudo-nMOS logic gates. Your gates receive three inputs, A, B, and C. Use a minimum number of transistors.

a) three-input NOR gate

b) three-input NAND gate

a) three-input AND gate

Exercise 1.65 *Resistor-Transistor Logic* (*RTL*) uses nMOS transistors to pull the gate output LOW and a weak resistor to pull the output HIGH when none of the paths to ground are active. A NOT gate built using RTL is shown in Figure 1.52. Sketch a three-input RTL NOR gate. Use a minimum number of transistors.

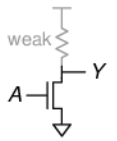


Figure 1.52 RTL NOT gate

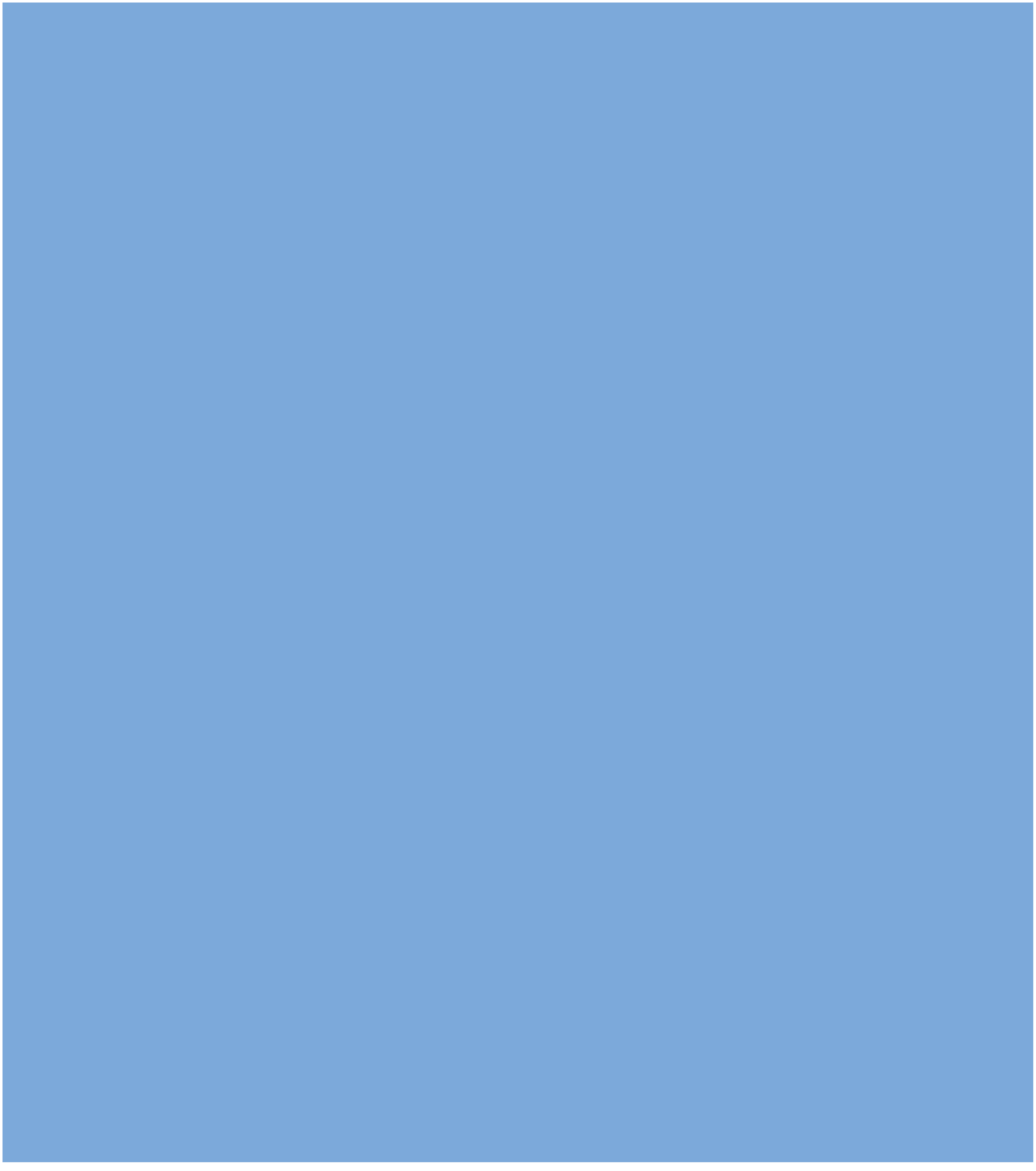
Interview Questions

These questions have been asked at interviews for digital design jobs.

Question 1.1 Sketch a transistor-level circuit for a CMOS four-input NOR gate.

Question 1.2 The king receives 64 gold coins in taxes but has reason to believe that one is counterfeit. He summons you to identify the fake coin. You have a balance that can hold coins on each side. How many times do you need to use the balance to find the lighter, fake coin?

Question 1.3 The professor, the teaching assistant, the digital design student, and the freshman track star need to cross a rickety bridge on a dark night. The bridge is so shakey that only two people can cross at a time. They have only one flashlight among them and the span is too long to throw the flashlight, so somebody must carry it back to the other people. The freshman track star can cross the bridge in 1 minute. The digital design student can cross the bridge in 2 minutes. The teaching assistant can cross the bridge in 5 minutes. The professor always gets distracted and takes 10 minutes to cross the bridge. What is the fastest time to get everyone across the bridge?



Combinational Logic Design

2

2.1 INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and outputs responding.

Peering inside the black box, circuits are composed of nodes and elements. An *element* is itself a circuit with inputs, outputs, and a specification. A *node* is a wire, whose voltage conveys a discrete-valued variable. Nodes are classified as *input*, *output*, or *internal*. Inputs receive values from the external world. Outputs deliver values to the external world. Wires that are not inputs or outputs are called internal nodes. Figure 2.2

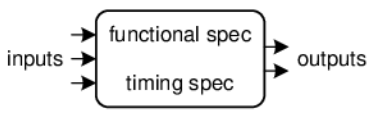


Figure 2.1 Circuit as a black box with inputs, outputs, and specifications

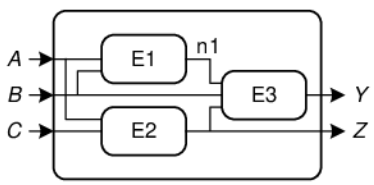


Figure 2.2 Elements and nodes

illustrates a circuit with three elements, E1, E2, and E3, and six nodes. Nodes *A*, *B*, and *C* are inputs. *Y* and Z are outputs. n1 is an internal node between E1 and E3.

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence. A combinational circuit is *memoryless*, but a sequential circuit has *memory*. This chapter focuses on combinational circuits, and Chapter 3 examines sequential circuits.

The functional specification of a combinational circuit expresses the output values in terms of the current input values. The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. We will initially concentrate on the functional specification, then return to the timing specification later in this chapter.

$Y = F(A, B) = A + B$

Figure 2.3 Combinational logic circuit

Figure 2.3 shows a combinational circuit with two inputs and one output. On the left of the figure are the inputs, *A* and *B*, and on the right is the output, *Y*. The symbol inside the box indicates that it is implemented using only combinational logic. In this example, the function, *F*, is specified to be OR: $Y = F(A, B) = A + B$. In words, we say the output, *Y*, is a function of the two inputs, *A* and *B*, namely $Y = A$ OR B.

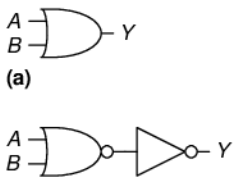
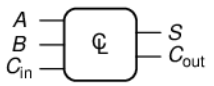


Figure 2.4 Two OR implementations

Figure 2.4 shows two possible *implementations* for the combinational logic circuit in Figure 2.3. As we will see repeatedly throughout the book, there are often many implementations for a single function. You choose which to use given the building blocks at your disposal and your design constraints. These constraints often include area, speed, power, and design time.



$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2.5 Multiple-output combinational circuit

Figure 2.5 shows a combinational circuit with multiple outputs. This particular combinational circuit is called a *full adder* and we will revisit it in Section 5.2.1. The two equations specify the function of the outputs, *S* and C_{out}, in terms of the inputs, *A*, *B*, and C_{in}.

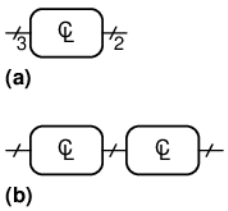


Figure 2.6 Slash notation for multiple signals

To simplify drawings, we often use a single line with a slash through it and a number next to it to indicate a *bus*, a bundle of multiple signals. The number specifies how many signals are in the bus. For example, Figure 2.6(a) represents a block of combinational logic with three inputs and two outputs. If the number of bits is unimportant or obvious from the context, the slash may be shown without a number. Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit

elements. A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

The rules of combinational composition are sufficient but not strictly necessary. Certain circuits that disobey these rules are still combinational, so long as the outputs depend only on the current values of the inputs. However, determining whether oddball circuits are combinational is more difficult, so we will usually restrict ourselves to combinational composition as a way to build combinational circuits.

Example 2.1 COMBINATIONAL CIRCUITS

Which of the circuits in Figure 2.7 are combinational circuits according to the rules of combinational composition?

Solution: Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

Large circuits such as microprocessors can be very complicated, so we use the principles from Chapter 1 to manage the complexity. Viewing a circuit as a black box with a well-defined interface and function is an

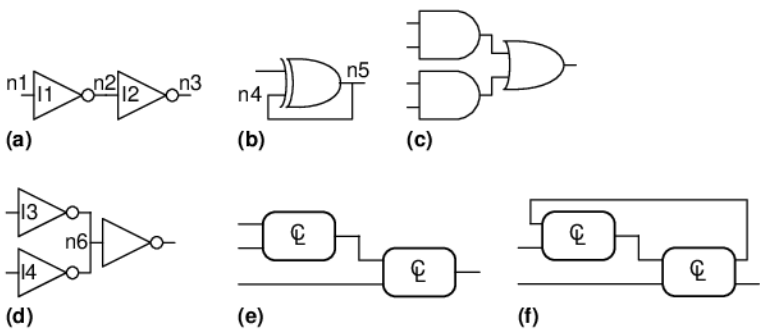


Figure 2.7 Example circuits

application of abstraction and modularity. Building the circuit out of smaller circuit elements is an application of hierarchy. The rules of combinational composition are an application of discipline.

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations. We show how to implement these equations using logic gates and how to analyze the speed of these circuits.

2.2 BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic. This section defines some terminology commonly used in Boolean equations, then shows how to write a Boolean equation for any logic function given its truth table.

2.2.1 Terminology

The *complement* of a variable, A, is its inverse, $\overline{A}$. The variable or its complement is called a *literal*. For example, A, $\overline{A}$, B, and $\overline{B}$ are literals. We call A the *true form* of the variable and $\overline{A}$ the *complementary form;* "true form" does not mean that A is TRUE, but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant*. $\overline{A}B$, $A\overline{B}\overline{C}$, and B are all implicants for a function of three variables. A *minterm* is a product involving all of the inputs to the function. $A\overline{B}\overline{C}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not, because it does not involve C. Similarly, the OR of one or more literals is called a *sum*. A *maxterm* is a sum involving all of the inputs to the function. $A + \overline{B} + C$ is a maxterm for a function of the three variables A, B, and C.

The *order of operations* is important when interpreting Boolean equations. Does $Y = A + BC$ mean $Y = (A \text{ OR } B) \text{ AND } C$ or $Y = A \text{ OR } (B \text{ AND } C)$? In Boolean equations, NOT has the highest *precedence*, followed by AND, then OR. Just as in ordinary equations, products are performed before sums. Therefore, the equation is read as $Y = A \text{ OR } (B \text{ AND } C)$. Equation 2.1 gives another example of order of operations.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \tag{2.1}$$

2.2.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs. Each row in a truth table is associated with a minterm

that is TRUE for that row. Figure 2.8 shows a truth table of two inputs, A and B. Each row shows its corresponding minterm. For example, the minterm for the first row is $\overline{A}\,\overline{B}$ because $\overline{A}\,\overline{B}$ is TRUE when $A = 0, B = 0$.

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. For example, in Figure 2.8, there is only one row (or minterm) for which the output Y is TRUE, shown circled in blue. Thus, $Y = \overline{A}B$. Figure 2.9 shows a truth table with more than one row in which the output is TRUE. Taking the sum of each of the circled minterms gives $Y = \overline{A}B + AB$.

This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms). Although there are many ways to write the same function, such as $Y = B\overline{A} + BA$, we will sort the minterms in the same order that they appear in the truth table, so that we always write the same Boolean expression for the same truth table.

A	B	Y	minterm
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}B$
1	0	0	$A\overline{B}$
1	1	0	AB

Figure 2.8 Truth table and minterms

Canonical form is just a fancy word for standard form. You can use the term to impress your friends and scare your enemies.

A	B	Y	minterm
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}B$
1	0	0	$A\overline{B}$
1	1	1	AB

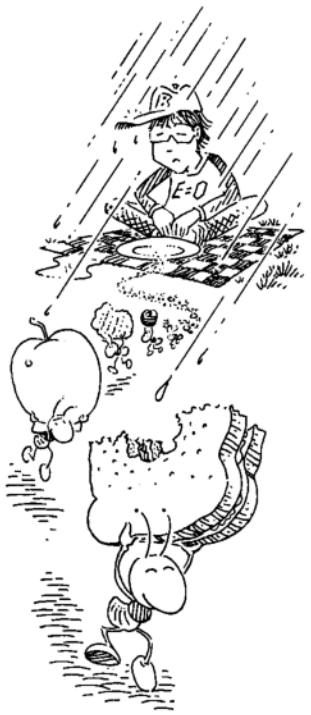
Figure 2.9 Truth table with multiple TRUE minterms

Example 2.2 SUM-OF-PRODUCTS FORM

Ben Bitdiddle is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE *only* if Ben enjoys the picnic.

Solution: First define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers. Figure 2.10 shows the truth table for Ben's picnic experience.

Using sum-of-products form, we write the equation as: $E = \overline{A}R$. We can build the equation using two inverters and a two-input AND gate, shown in Figure 2.11(a). You may recognize this truth table as the NOR function from Section 1.5.5: $E = A$ NOR $R = \overline{A + R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}R$ and $\overline{A + R}$, are equivalent.



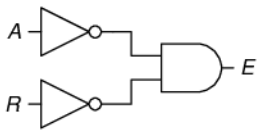
The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Figure 2.12 shows a random three-input truth table. The sum-of-products form of the logic function is

$$Y = \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C \tag{2.2}$$

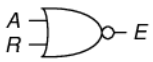
Unfortunately, sum-of-products form does not necessarily generate the simplest equation. In Section 2.3 we show how to write the same function using fewer terms.

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.10 Ben's truth table



(a)



(b)

Figure 2.11 Ben's circuit

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.12 Random three-input truth table

A	B	Y	maxterm
0	0	0	A + B
0	1	1	A + $\overline{B}$
1	0	0	$\overline{A}$ + B
1	1	1	$\overline{A}$ + $\overline{B}$

Figure 2.13 Truth table with multiple FALSE maxterms

2.2.3 Product-of-Sums Form

An alternative way of expressing Boolean functions is the *product-of-sums canonical form*. Each row of a truth table corresponds to a maxterm that is FALSE for that row. For example, the maxterm for the first row of a two-input truth table is $(A + B)$ because $(A + B)$ is FALSE when $A = 0$, $B = 0$. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE.

Example 2.3 PRODUCT-OF-SUMS FORM

Write an equation in product-of-sums form for the truth table in Figure 2.13.

Solution: The truth table has two rows in which the output is FALSE. Hence, the function can be written in product-of-sums form as $Y = (A + B)(\overline{A} + B)$. The first maxterm, $(A + B)$, guarantees that $Y = 0$ for $A = 0$, $B = 0$, because any value AND 0 is 0. Likewise, the second maxterm, $(\overline{A} + B)$, guarantees that $Y = 0$ for $A = 1$, $B = 0$. Figure 2.13 is the same truth table as Figure 2.9, showing that the same function can be written in more than one way.

Similarly, a Boolean equation for Ben's picnic from Figure 2.10 can be written in product-of-sums form by circling the three rows of 0's to obtain $E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$. This is uglier than the sum-of-products equation, $E = \overline{A}\overline{R}$, but the two equations are logically equivalent.

Sum-of-products produces the shortest equations when the output is TRUE on only a few rows of a truth table; product-of-sums is simpler when the output is FALSE on only a few rows of a truth table.

2.3 BOOLEAN ALGEBRA

In the previous section, we learned how to write a Boolean expression given a truth table. However, that expression does not necessarily lead to the simplest set of logic gates. Just as you use algebra to simplify mathematical equations, you can use *Boolean algebra* to simplify Boolean equations. The rules of Boolean algebra are much like those of ordinary algebra but are in some cases simpler, because variables have only two possible values: 0 or 1.

Boolean algebra is based on a set of axioms that we assume are correct. Axioms are unprovable in the sense that a definition cannot be proved. From these axioms, we prove all the theorems of Boolean algebra. These theorems have great practical significance, because they teach us how to simplify logic to produce smaller and less costly circuits.

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators • (AND) and + (OR) are interchanged, the statement will still be correct. We use the prime (′) symbol to denote the *dual* of a statement.

2.3.1 Axioms

Table 2.1 states the axioms of Boolean algebra. These five axioms and their duals define Boolean variables and the meanings of NOT, AND, and OR. Axiom A1 states that a Boolean variable B is 0 if it is not 1. The axiom's dual, A1′, states that the variable is 1 if it is not 0. Together, A1 and A1′ tell us that we are working in a Boolean or binary field of 0's and 1's. Axioms A2 and A2′ define the NOT operation. Axioms A3 to A5 define AND; their duals, A3′ to A5′ define OR.

2.3.2 Theorems of One Variable

Theorems T1 to T5 in Table 2.2 describe how to simplify equations involving one variable.

The *identity* theorem, T1, states that for any Boolean variable B, B AND $1 = B$. Its dual states that B OR $0 = B$. In hardware, as shown in Figure 2.14, T1 means that if one input of a two-input AND gate is always 1, we can remove the AND gate and replace it with a wire connected to the variable input (B). Likewise, T1′ means that if one input of a two-input OR gate is always 0, we can replace the OR gate with a wire connected to B. In general, gates cost money, power, and delay, so replacing a gate with a wire is beneficial.

The *null element theorem*, T2, says that B AND 0 is always equal to 0. Therefore, 0 is called the *null* element for the AND operation, because it nullifies the effect of any other input. The dual states that B OR 1 is always equal to 1. Hence, 1 is the null element for the OR operation. In hardware, as shown in Figure 2.15, if one input of an AND gate is 0, we can replace the AND gate with a wire that is tied

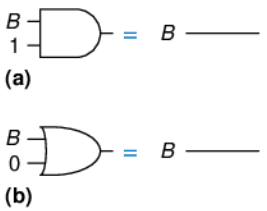


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1′

The null element theorem leads to some outlandish statements that are actually true! It is particularly dangerous when left in the hands of advertisers: YOU WILL GET A MILLION DOLLARS or we'll send you a toothbrush in the mail. (You'll most likely be receiving a toothbrush in the mail.)

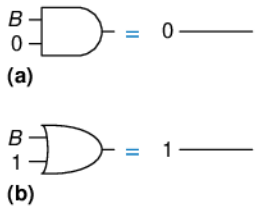


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

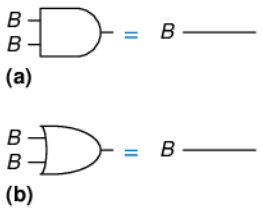


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

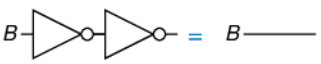


Figure 2.17 Involution theorem in hardware: T4

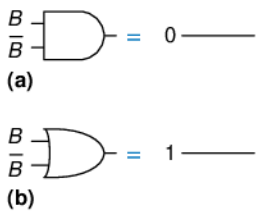


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

LOW (to 0). Likewise, if one input of an OR gate is 1, we can replace the OR gate with a wire that is tied HIGH (to 1).

Idempotency, T3, says that a variable AND itself is equal to just itself. Likewise, a variable OR itself is equal to itself. The theorem gets its name from the Latin roots: *idem* (same) and *potent* (power). The operations return the same thing you put into them. Figure 2.16 shows that idempotency again permits replacing a gate with a wire.

Involution, T4, is a fancy way of saying that complementing a variable twice results in the original variable. In digital electronics, two wrongs make a right. Two inverters in series logically cancel each other out and are logically equivalent to a wire, as shown in Figure 2.17. The dual of T4 is itself.

The *complement theorem,* T5 (Figure 2.18), states that a variable AND its complement is 0 (because one of them has to be 0). And, by duality, a variable OR its complement is 1 (because one of them has to be 1).

2.3.3 Theorems of Several Variables

Theorems T6 to T12 in Table 2.3 describe how to simplify equations involving more than one Boolean variable.

Commutativity and *associativity,* T6 and T7, work the same as in traditional algebra. By commutativity, the *order* of inputs for an AND or OR function does not affect the value of the output. By associativity, the specific groupings of inputs do not affect the value of the output.

The *distributivity theorem,* T8, is the same as in traditional algebra, but its dual, T8′, is not. By T8, AND distributes over OR, and by T8′, OR distributes over AND. In traditional algebra, multiplication distributes over addition but addition does not distribute over multiplication, so that $(B + C) \times (B + D) \neq B + (C \times D)$.

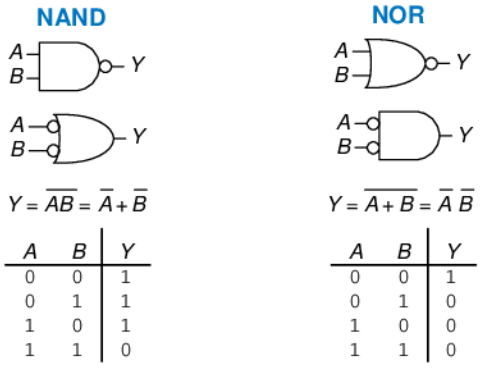
The *covering, combining,* and *consensus* theorems, T9 to T11, permit us to eliminate redundant variables. With some thought, you should be able to convince yourself that these theorems are correct.

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

De Morgan's Theorem, T12, is a particularly powerful tool in digital design. The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

According to De Morgan's theorem, a NAND gate is equivalent to an OR gate with inverted inputs. Similarly, a NOR gate is equivalent to an AND gate with inverted inputs. Figure 2.19 shows these *De Morgan equivalent gates* for NAND and NOR gates. The two symbols shown for each function are called *duals*. They are logically equivalent and can be used interchangeably.



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.19 De Morgan equivalent gates

Augustus De Morgan, died 1871. A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2."

The inversion circle is called a *bubble*. Intuitively, you can imagine that "pushing" a bubble through the gate causes it to come out at the other side and flips the body of the gate from AND to OR or vice versa. For example, the NAND gate in Figure 2.19 consists of an AND body with a bubble on the output. Pushing the bubble to the left results in an OR body with bubbles on the inputs. The underlying rules for bubble pushing are

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on *all* gate inputs forward toward the output puts a bubble on the output.

Section 2.5.2 uses bubble pushing to help analyze circuits.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

FIGURE 2.20 Truth table showing Y and $\overline{Y}$

A	B	Y	$\overline{Y}$	minterm
0	0	0	1	$\overline{A}\,\overline{B}$
0	1	0	1	$\overline{A}\,B$
1	0	1	0	$A\,\overline{B}$
1	1	1	0	$A\,B$

Figure 2.21 Truth table showing minterms for $\overline{Y}$

Example 2.4 DERIVE THE PRODUCT-OF-SUMS FORM

Figure 2.20 shows the truth table for a Boolean function, Y, and its complement, $\overline{Y}$. Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

Solution: Figure 2.21 shows the minterms (circled) contained in $\overline{Y}$. The sum-of-products canonical form of $\overline{Y}$ is

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B \tag{2.3}$$

Taking the complement of both sides and applying De Morgan's Theorem twice, we get:

$$\overline{\overline{Y}} = Y = \overline{\overline{A}\,\overline{B} + \overline{A}B} = (\overline{\overline{A}\,\overline{B}})(\overline{\overline{A}B}) = (A + B)(A + \overline{B}) \tag{2.4}$$

2.3.4 The Truth Behind It All

The curious reader might wonder how to prove that a theorem is true. In Boolean algebra, proofs of theorems with a finite number of variables are easy: just show that the theorem holds for all possible values of these variables. This method is called *perfect induction* and can be done with a truth table.

Example 2.5 PROVING THE CONSENSUS THEOREM

Prove the consensus theorem, T11, from Table 2.3.

Solution: Check both sides of the equation for all eight combinations of B, C, and D. The truth table in Figure 2.22 illustrates these combinations. Because $BC + \overline{B}D + CD = BC + \overline{B}D$ for all cases, the theorem is proved.

B	C	D	$BC + \overline{B}D + CD$	$BC + \overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Figure 2.22 Truth table proving T11

2.3.5 Simplifying Equations

The theorems of Boolean algebra help us simplify Boolean equations. For example, consider the sum-of-products expression from the truth table of Figure 2.9: $Y = \overline{A}\overline{B} + A\overline{B}$. By Theorem T10, the equation simplifies to $Y = \overline{B}$. This may have been obvious looking at the truth table. In general, multiple steps may be necessary to simplify more complex equations.

The basic principle of simplifying sum-of-products equations is to combine terms using the relationship $PA + P\overline{A} = P$, where P may be any implicant. How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants to form a new implicant with fewer literals. The implicants in a minimal equation must all be prime implicants. Otherwise, they could be combined to reduce the number of literals.

Example 2.6 EQUATION MINIMIZATION

Minimize Equation 2.2: $\overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$.

Solution: We start with the original equation and apply Boolean theorems step by step, as shown in Table 2.4.

Have we simplified the equation completely at this point? Let's take a closer look. From the original equation, the minterms $\overline{A}\overline{B}\overline{C}$ and $A\overline{B}\overline{C}$ differ only in the variable A. So we combined the minterms to form $\overline{B}\overline{C}$. However, if we look at the original equation, we note that the last two minterms $A\overline{B}\overline{C}$ and $A\overline{B}C$ also differ by a single literal (C and $\overline{C}$). Thus, using the same method, we could have combined these two minterms to form the minterm $A\overline{B}$. We say that implicants $\overline{B}\overline{C}$ and $A\overline{B}$ *share* the minterm $A\overline{B}\overline{C}$.

So, are we stuck with simplifying only one of the minterm pairs, or can we simplify both? Using the idempotency theorem, we can duplicate terms as many times as we want: $B = B + B + B + B \ldots$. Using this principle, we simplify the equation completely to its two prime implicants, $\overline{B}\overline{C} + A\overline{B}$, as shown in Table 2.5.

Table 2.4 Equation minimization

Step	Equation	Justification
	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	
1	$\overline{B}\overline{C}\,(\overline{A} + A) + A\overline{B}C$	T8: Distributivity
2	$\overline{B}\overline{C}\,(1) + A\overline{B}C$	T5: Complements
3	$\overline{B}\overline{C} + A\overline{B}C$	T1: Identity

Table 2.5 Improved equation minimization

Step	Equation	Justification
	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	
1	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	T3: Idempotency
2	$\overline{B}\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: Distributivity
3	$\overline{B}\overline{C}(1) + A\overline{B}(1)$	T5: Complements
4	$\overline{B}\overline{C} + A\overline{B}$	T1: Identity

Although it is a bit counterintuitive, *expanding* an implicant (for example, turning AB into $ABC + AB\overline{C}$) is sometimes useful in minimizing equations. By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called Karnaugh maps that makes the process easier.

Why bother simplifying a Boolean equation if it remains logically equivalent? Simplifying reduces the number of gates used to physically implement the function, thus making it smaller, cheaper, and possibly faster. The next section describes how to implement Boolean equations with logic gates.

2.4 FROM LOGIC TO GATES

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together. For example, the schematic in Figure 2.23 shows a possible hardware implementation of our favorite logic function, Equation 2.2:

$$Y = \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C.$$

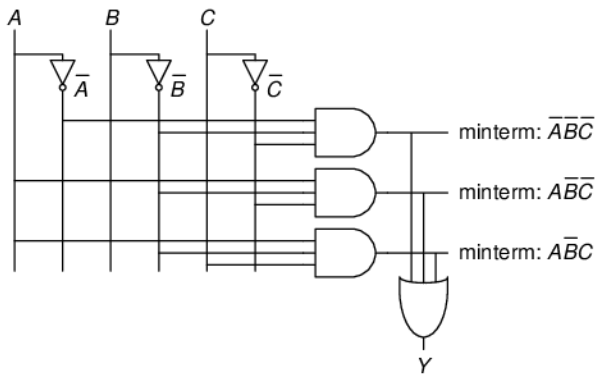


Figure 2.23 Schematic of $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking of what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

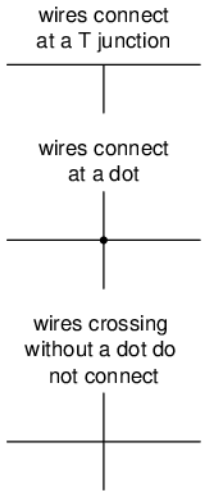


Figure 2.24 Wire connections

The last three guidelines are illustrated in Figure 2.24.

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to Figure 2.23. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This style is called a *programmable logic array (PLA)* because the inverters, AND gates, and OR gates are arrayed in a systematic fashion. PLAs will be discussed further in Section 5.6.

Figure 2.25 shows an implementation of the simplified equation we found using Boolean algebra in Example 2.6. Notice that the simplified circuit has significantly less hardware than that of Figure 2.23. It may also be faster, because it uses gates with fewer inputs.

We can reduce the number of gates even further (albeit by a single inverter) by taking advantage of inverting gates. Observe that $\overline{BC}$ is an

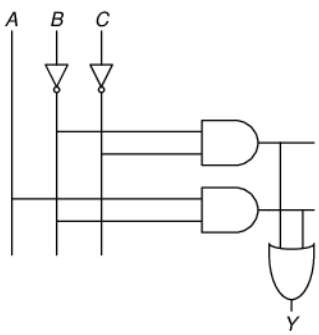


Figure 2.25 Schematic of $Y = \overline{B\overline{C}} + A\overline{B}$

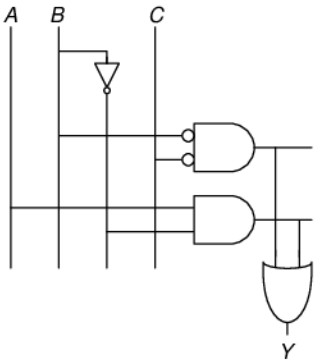


Figure 2.26 Schematic using fewer gates

AND with inverted inputs. Figure 2.26 shows a schematic using this optimization to eliminate the inverter on C. Recall that by De Morgan's theorem the AND with inverted inputs is equivalent to a NOR gate. Depending on the implementation technology, it may be cheaper to use the fewest gates or to use certain types of gates in preference to others. For example, NANDs and NORs are preferred over ANDs and ORs in CMOS implementations.

Many circuits have multiple outputs, each of which computes a separate Boolean function of the inputs. We can write a separate truth table for each output, but it is often convenient to write all of the outputs on a single truth table and sketch one schematic with all of the outputs.

Example 2.7 MULTIPLE-OUTPUT CIRCUITS

The dean, the department chair, the teaching assistant, and the dorm social chair each use the auditorium from time to time. Unfortunately, they occasionally conflict, leading to disasters such as the one that occurred when the dean's fundraising meeting with crusty trustees happened at the same time as the dorm's BTB[1] party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, $A_3, \ldots, A_0$, and four outputs, $Y_3, \ldots, Y_0$. These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts her input when she requests the auditorium for the next day. The system asserts at most one output, granting the auditorium to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority.

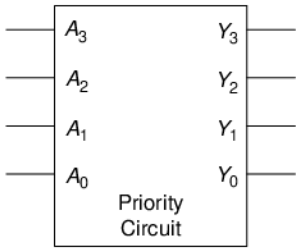
Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.

Solution: This function is called a four-input *priority circuit*. Its symbol and truth table are shown in Figure 2.27.

We could write each output in sum-of-products form and reduce the equations using Boolean algebra. However, the simplified equations are clear by inspection from the functional description (and the truth table): Y_3 is TRUE whenever A_3 is asserted, so $Y_3 = A_3$. Y_2 is TRUE if A_2 is asserted and A_3 is not asserted, so $Y_2 = \overline{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher priority inputs is asserted: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$. And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$. The schematic is shown in Figure 2.28. An experienced designer can often implement a logic circuit by inspection. Given a clear specification, simply turn the words into equations and the equations into gates.

---

[1] Black light, twinkies, and beer.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2.27 Priority circuit

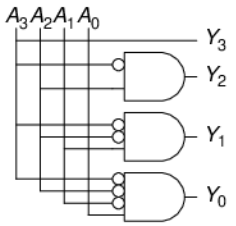


Figure 2.28 Priority circuit schematic

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)

Notice that if A_3 is asserted in the priority circuit, the outputs *don't care* what the other inputs are. We use the symbol X to describe inputs that the output doesn't care about. Figure 2.29 shows that the four-input priority circuit truth table becomes much smaller with don't cares. From this truth table, we can easily read the Boolean equations in sum-of-products form by ignoring inputs with X's. Don't cares can also appear in truth table outputs, as we will see in Section 2.7.3.

X is an overloaded symbol that means "don't care" in truth tables and "contention" in logic simulation (see Section 2.6.1). Think about the context so you don't mix up the meanings. Some authors use D or ? instead for "don't care" to avoid this ambiguity.

2.5 MULTILEVEL COMBINATIONAL LOGIC

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of

OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts. Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

2.5.1 Hardware Reduction

Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. For example consider building a three-input XOR using the two-level techniques we have studied so far.

Recall that an N-input XOR produces a TRUE output when an odd number of inputs are TRUE. Figure 2.30 shows the truth table for a three-input XOR with the rows circled that produce TRUE outputs. From the truth table, we read off a Boolean equation in sum-of-products form in Equation 2.5. Unfortunately, there is no way to simplify this equation into fewer implicants.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{2.5}$$

On the other hand, $A \oplus B \oplus C = (A \oplus B) \oplus C$ (prove this to yourself by perfect induction if you are in doubt). Therefore, the three-input XOR can be built out of a cascade of two-input XORs, as shown in Figure 2.31.

Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two-level sum-of-products implementation. A much better option is to use a tree of two-input XOR gates, as shown in Figure 2.32.

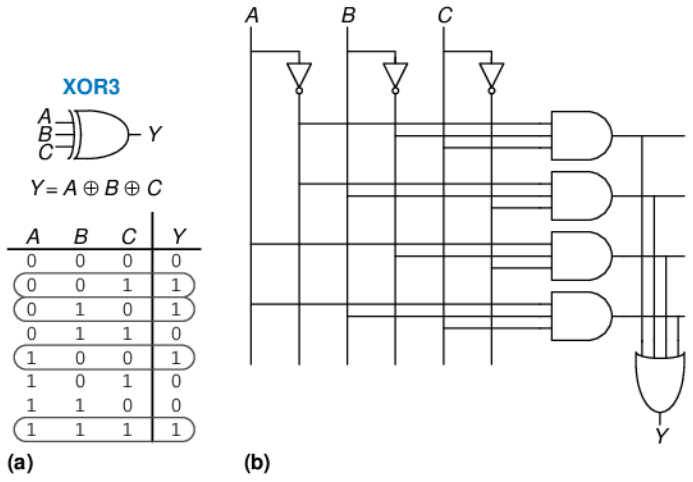


Figure 2.30 Three-input XOR: (a) functional specification and (b) two-level logic implementation

Selecting the best multilevel implementation of a specific logic function is not a simple process. Moreover, "best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption. In Chapter 5, you will see that the "best" circuit in one technology is not necessarily the best in another. For example, we have been using ANDs and ORs, but in CMOS, NANDs and NORs are more efficient. With some experience, you will find that you can create a good multilevel design by inspection for most circuits. You will develop some of this experience as you study circuit examples through the rest of this book. As you are learning, explore various design options and think about the trade-offs. Computer-aided design (CAD) tools are also available to search a vast space of possible multilevel designs and seek the one that best fits your constraints given the available building blocks.

Figure 2.31 Three-input XOR using two two-input XORs

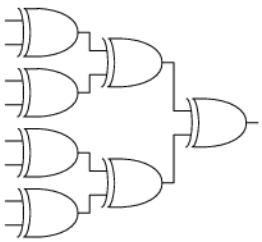


Figure 2.32 Eight-input XOR using seven two-input XORs

2.5.2 Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits prefer NANDs and NORs over ANDs and ORs. But reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Figure 2.33 shows a multilevel circuit whose function is not immediately clear by inspection. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined. Building on the principles from Section 2.3.3, the guidelines for bubble pushing are as follows:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (for example, Y) instead of the complement of the output ($\overline{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

Figure 2.34 shows how to redraw Figure 2.33 according to the bubble pushing guidelines. Starting at the output, Y, the NAND gate has a bubble on the output that we wish to eliminate. We push the output bubble back to form an OR with inverted inputs, shown in

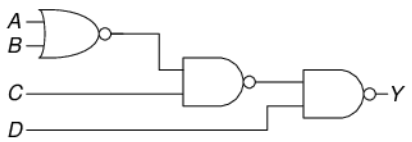


Figure 2.33 Multilevel circuit using NANDs and NORs

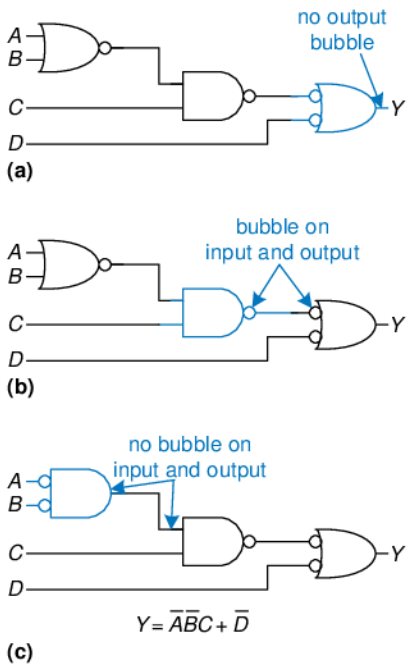


Figure 2.34 Bubble-pushed circuit

Figure 2.34(a). Working to the left, the rightmost gate has an input bubble that cancels with the output bubble of the middle NAND gate, so no change is necessary, as shown in Figure 2.34(b). The middle gate has no input bubble, so we transform the leftmost gate to have no output bubble, as shown in Figure 2.34(c). Now all of the bubbles in the circuit cancel except at the inputs, so the function can be read by inspection in terms of ANDs and ORs of true or complementary inputs: $Y = \overline{A}\overline{B}C + \overline{D}$.

For emphasis of this last point, Figure 2.35 shows a circuit logically equivalent to the one in Figure 2.34. The functions of internal nodes are labeled in blue. Because bubbles in series cancel, we can ignore the bubble on the output of the middle gate and the input of the rightmost gate to produce the logically equivalent circuit of Figure 2.35.

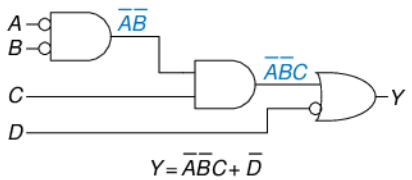


Figure 2.35 Logically equivalent bubble-pushed circuit

Example 2.8 BUBBLE PUSHING FOR CMOS LOGIC

Most designers think in terms of AND and OR gates, but suppose you would like to implement the circuit in Figure 2.36 in CMOS logic, which favors NAND and NOR gates. Use bubble pushing to convert the circuit to NANDs, NORs, and inverters.

Solution: A brute force solution is to just replace each AND gate with a NAND and an inverter, and each OR gate with a NOR and an inverter, as shown in Figure 2.37. This requires eight gates. Notice that the inverter is drawn with the bubble on the front rather than back, to emphasize how the bubble can cancel with the preceding inverting gate.

For a better solution, observe that bubbles can be added to the output of a gate and the input of the next gate without changing the function, as shown in Figure 2.38(a). The final AND is converted to a NAND and an inverter, as shown in Figure 2.38(b). This solution requires only five gates.

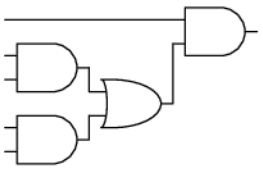


Figure 2.36 Circuit using ANDs and ORs

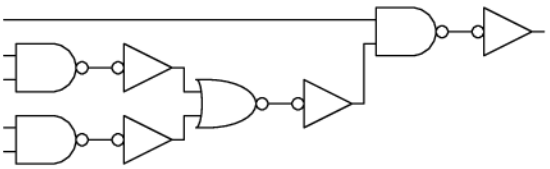


Figure 2.37 Poor circuit using NANDs and NORs

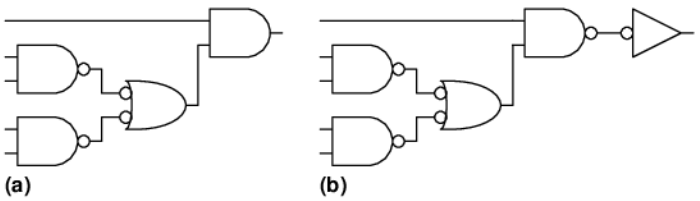


Figure 2.38 Better circuit using NANDs and NORs

2.6 X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

2.6.1 Illegal Value: X

The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.39 shows a case where node *Y* is driven both HIGH and LOW. This situation, called *contention*, is considered to be an error

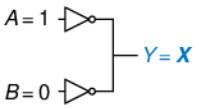


Figure 2.39 Circuit with contention

and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone. Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.

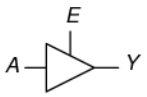
X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume it is an X to warn you of the problem.

As mentioned in Section 2.4, digital designers also use the symbol X to indicate "don't care" values in truth tables. Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant. When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

2.6.2 Floating Value: Z

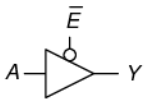
The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high Z*. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system. A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.

One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

The *tristate buffer*, shown in Figure 2.40, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input, *A*, an output, *Y*, and an *enable*, *E*. When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

The tristate buffer in Figure 2.40 has an *active high* enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure 2.41 shows a tristate buffer with an *active low* enable. When the enable is LOW (0),

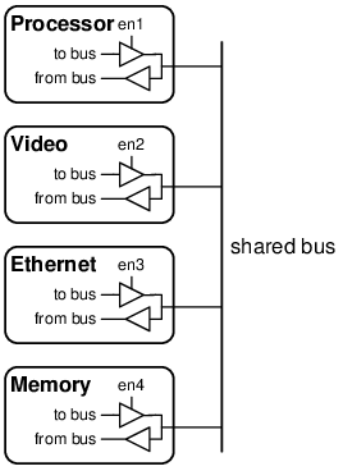


Figure 2.42 Tristate bus connecting multiple chips

the buffer is enabled. We show that the signal is active low by putting a bubble on its input wire. We often indicate an active low input by drawing a bar over its name, $\overline{E}$, or appending the word "bar" after its name, *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in Figure 2.42. Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time. Such tristate busses were once common. However, in modern computers, higher speeds are possible with *point-to-point links*, in which chips are connected to each other directly rather than over a shared bus.

2.7 KARNAUGH MAPS

After working through several minimizations of Boolean equations using Boolean algebra, you will realize that, if you're not careful, you sometimes end up with a completely *different* equation instead of a simplified equation. *Karnaugh maps* (*K-maps*) are a graphical method for simplifying Boolean equations. They were invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs. K-maps work well for problems with up to four variables. More important, they give insight into manipulating Boolean equations.

Maurice Karnaugh, 1924–. Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952. Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

Gray codes were patented (U.S. Patent 2,632,058) by Frank Gray, a Bell Labs researcher, in 1953. They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit.

Gray codes generalize to any number of bits. For example, a 3-bit Gray code sequence is:

```
000, 001, 011, 010,
110, 111, 101, 100
```

Lewis Carroll posed a related puzzle in *Vanity Fair* in 1879.

> "The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the puzzle consists of linking these together by interposing other words, each of which shall differ from the next word in one letter only. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word."

For example, SHIP to DOCK:

```
SHIP, SLIP, SLOP,
SLOT, SOOT, LOOT,
LOOK, LOCK, DOCK.
```

Can you find a shorter sequence?

Recall that logic minimization involves combining terms. Two terms containing an implicant, P, and the true and complementary forms of some variable, A, are combined to eliminate A: $PA + P\overline{A} = P$. Karnaugh maps make these combinable terms easy to see by putting them next to each other in a grid.

Figure 2.43 shows the truth table and K-map for a three-input function. The top row of the K-map gives the four possible values for the A and B inputs. The left column gives the two possible values for the C input. Each square in the K-map corresponds to a row in the truth table and contains the value of the output, Y, for that row. For example, the top left square corresponds to the first row in the truth table and indicates that the output value $Y = 1$ when $ABC = 000$. Just like each row in a truth table, each square in a K-map represents a single minterm. For the purpose of explanation, Figure 2.43(c) shows the minterm corresponding to each square in the K-map.

Each square, or minterm, differs from an adjacent square by a change in a single variable. This means that adjacent squares share all the same literals except one, which appears in true form in one square and in complementary form in the other. For example, the squares representing the minterms $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$ are adjacent and differ only in the variable C. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10. This order is called a *Gray code*. It differs from ordinary binary order (00, 01, 10, 11) in that adjacent entries differ only in a single variable. For example, 01 : 11 only changes A from 0 to 1, while 01 : 10 would change A from 1 to 0 and B from 0 to 1. Hence, writing the combinations in binary order would not have produced our desired property of adjacent squares differing only in one variable.

The K-map also "wraps around." The squares on the far right are effectively adjacent to the squares on the far left, in that they differ only in one variable, A. In other words, you could take the map and roll it into a cylinder, then join the ends of the cylinder to form a torus (i.e., a donut), and still guarantee that adjacent squares would differ only in one variable.

2.7.1 Circular Thinking

In the K-map in Figure 2.43, only two minterms are present in the equation, $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$, as indicated by the 1's in the left column. Reading the minterms from the K-map is exactly equivalent to reading equations in sum-of-products form directly from the truth table.

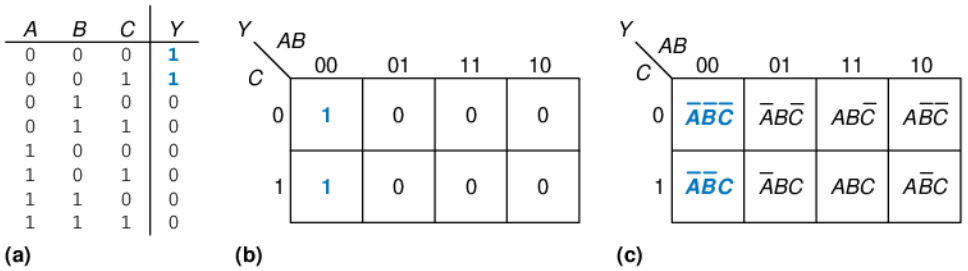


Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms

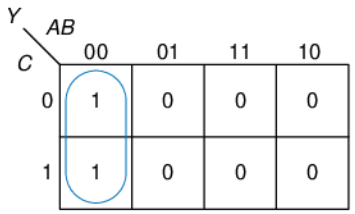


Figure 2.44 K-map minimization

As before, we can use Boolean algebra to minimize equations in sum-of-products form.

$$Y = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C = \overline{A}\overline{B}(\overline{C} + C) = \overline{A}\overline{B} \tag{2.6}$$

K-maps help us do this simplification graphically by *circling* 1's in adjacent squares, as shown in Figure 2.44. For each circle, we write the corresponding implicant. Remember from Section 2.2 that an implicant is the product of one or more literals. Variables whose true *and* complementary forms are both in the circle are excluded from the implicant. In this case, the variable C has both its true form (1) and its complementary form (0) in the circle, so we do not include it in the implicant. In other words, Y is TRUE when $A = B = 0$, independent of C. So the implicant is $\overline{A}\overline{B}$. This K-map gives the same answer we reached using Boolean algebra.

2.7.2 Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as large as possible. Then read off the implicants that were circled.

More formally, recall that a Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants. Each circle on the K-map represents an implicant. The largest possible circles are prime implicants.

For example, in the K-map of Figure 2.44, $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$ are implicants, but *not* prime implicants. Only $\overline{A}\overline{B}$ is a prime implicant in that K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

Example 2.9 MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP

Suppose we have the function $Y = F(A, B, C)$ with the K-map shown in Figure 2.45. Minimize the equation using the K-map.

Solution: Circle the 1's in the K-map using as few circles as possible, as shown in Figure 2.46. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two (2×1 and 2×2). We form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.

For example, in the 2×1 circle, the true and complementary forms of B are included in the circle, so we *do not* include B in the prime implicant. However, only the true form of $A(A)$ and complementary form of $C(\overline{C})$ are in this circle, so we include these variables in the prime implicant $A\overline{C}$. Similarly, the 2×2 circle covers all squares where $B = 0$, so the prime implicant is $\overline{B}$.

Notice how the top-right square (minterm) is covered twice to make the prime implicant circles as large as possible. As we saw with Boolean algebra techniques, this is equivalent to sharing a minterm to reduce the size of the

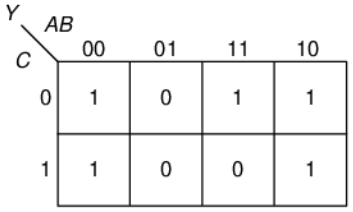


Figure 2.45 K-map for Example 2.9

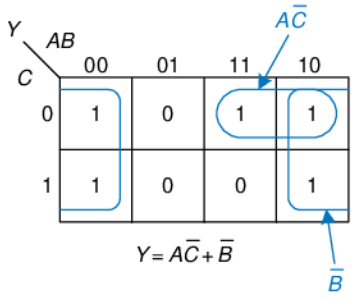


Figure 2.46 Solution for Example 2.9

implicant. Also notice how the circle covering four squares wraps around the sides of the K-map.

Example 2.10 SEVEN-SEGMENT DISPLAY DECODER

A *seven-segment display decoder* takes a 4-bit data input, $D_{3:0}$, and produces seven outputs to control light-emitting diodes to display a digit from 0 to 9. The seven outputs are often called segments *a* through *g*, or S_a–S_g, as defined in Figure 2.47. The digits are shown in Figure 2.48. Write a truth table for the outputs, and use K-maps to find Boolean equations for outputs S_a and S_b. Assume that illegal input values (10–15) produce a blank readout.

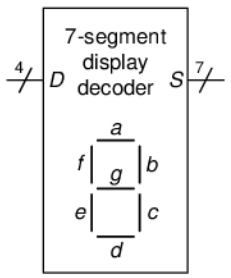


Figure 2.47 Seven-segment display decoder icon

Solution: The truth table is given in Table 2.6. For example, an input of 0000 should turn on all segments except S_g.

Each of the seven outputs is an independent function of four variables. The K-maps for outputs S_a and S_b are shown in Figure 2.49. Remember that adjacent squares may differ in only a single variable, so we label the rows and columns in Gray code order: 00, 01, 11, 10. Be careful to also remember this ordering when *entering* the output values into the squares.

Next, circle the prime implicants. Use the fewest number of circles necessary to cover all the 1's. A circle can wrap around the edges (vertical *and* horizontal), and a 1 may be circled more than once. Figure 2.50 shows the prime implicants and the simplified Boolean equations.

Note that the minimal set of prime implicants is not unique. For example, the 0000 entry in the S_a K-map was circled along with the 1000 entry to produce the $\overline{D}_2\overline{D}_1\overline{D}_0$ minterm. The circle could have included the 0010 entry instead, producing a $\overline{D}_3\overline{D}_2\overline{D}_0$ minterm, as shown with dashed lines in Figure 2.51.

Figure 2.52 illustrates (see page 78) a common error in which a nonprime implicant was chosen to cover the 1 in the upper left corner. This minterm, $\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$, gives a sum-of-products equation that is *not* minimal. The minterm could have been combined with either of the adjacent ones to form a larger circle, as was done in the previous two figures.

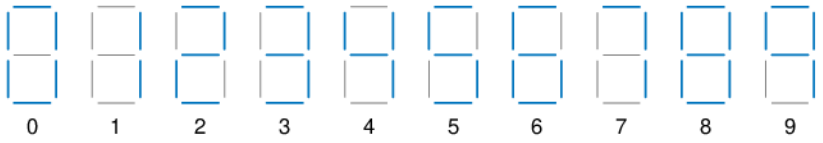


Figure 2.48 Seven-segment display digits

Table 2.6 Seven-segment display decoder truth table

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

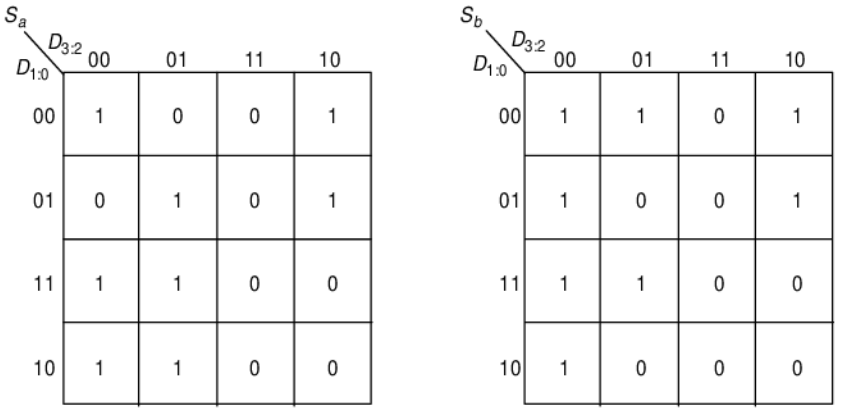


Figure 2.49 Karnaugh maps for S_a and S_b

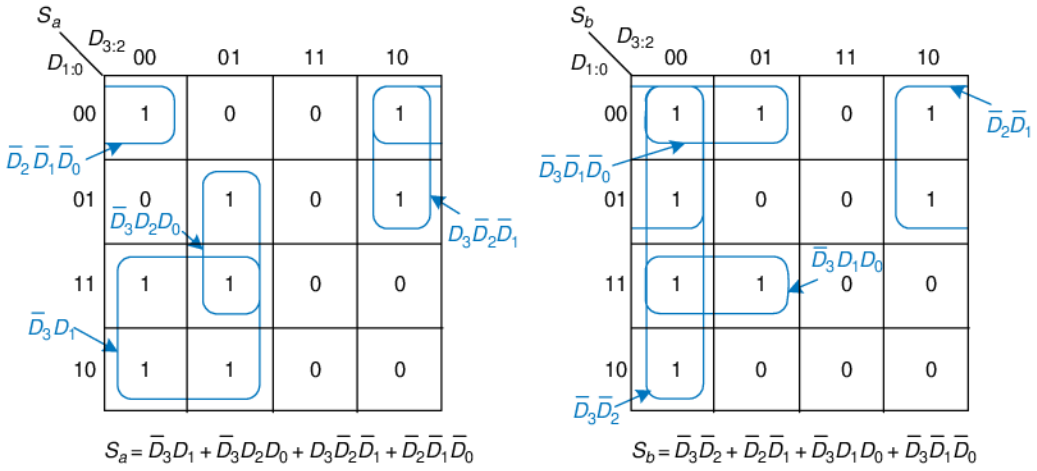


Figure 2.50 K-map solution for Example 2.10

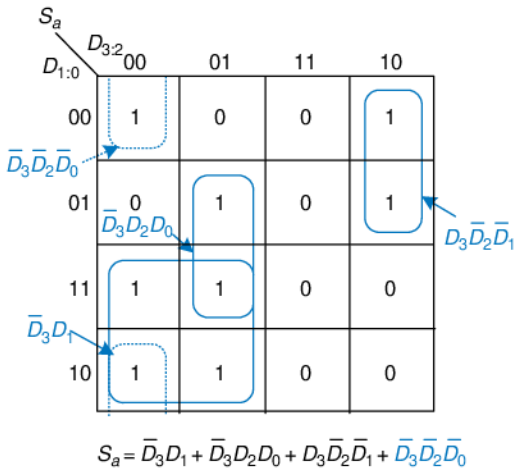


Figure 2.51 Alternative K-map for S_a showing different set of prime implicants

2.7.3 Don't Cares

Recall that "don't care" entries for truth table inputs were introduced in Section 2.4 to reduce the number of rows in the table when some variables do not affect the output. They are indicated by the symbol X, which means that the entry can be either 0 or 1.

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.