

CISC 260 Machine Organization and Assembly Language

Practice Midterm Exam

This is an open-note exam. You are allowed to use notes. You are NOT allowed to use electronic devices except standard calculators.

1. [25 points] Data representations and arithmetics

a. Convert 33_{ten} into a 8-bit two's complement binary number.

Answer: 0010 0001

b. What decimal number does the following two's complement 8-bit binary number represent?

$$1100\ 1010 = -54_{\text{ten}}$$

c. Is there an overflow for an 8-bit machine when subtracting a two's complement integer x from a two's complement integer y as given below? Show your work.

$$x = 1000\ 1011 \text{ and } y = 0111\ 0100$$

Answer:

X is negative and y is positive. Therefore, y-x is adding two positive integers, where overflow occurs when the result is negative.

$$-x = 0111\ 0101$$

$$0111\ 0100\ (y)$$

$$0111\ 0101\ (x)$$

$$1110\ 1001\ (y-x)$$

Therefore, there is an overflow.

d. Show the negation of the following integer in two's complement.

$$X = 1101\ 0110\ 0111\ 0101_{\text{two}}$$

Answer: $-x = 0010\ 1001\ 1000\ 1011_{\text{two}}$

e. In multiplying the following two integers A and B , how many times the (properly shifted) multiplicand is added to the (intermediate) product $C = A \times B$ if the multiplication is implemented using the shift-add algorithm?

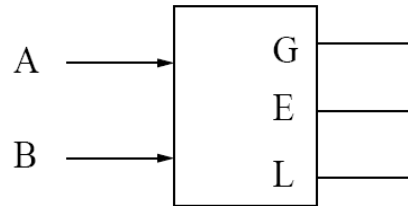
A = 1010 0101

B = 0110 1001

Answer: 4.

2. [20 points] Boolean Logic and Gates

A comparator circuit has two 1 bit inputs A and B and three 1 bit outputs G (greater), E (Equal) and L (less than)



$G = 1$, if $A > B$
0, otherwise

$E = 1$, if $A = B$
0, otherwise

$L = 1$, if $A < B$
0, otherwise

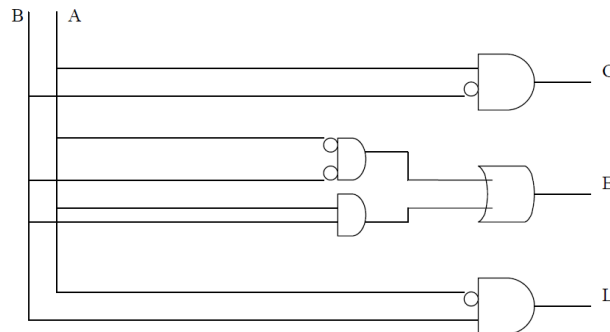
a. Fill out the truth table

A	B	G	E	L
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

b. Write the Boolean expression in canonical form corresponding to the above truth table

$$G = A \& \sim B; E = (\sim A \& \sim B) \mid (A \& B); L = \sim A \& B$$

c. Implement the circuit by using AND, OR and NOT gates. Draw the wiring diagram.



3. [25 points] ARM Instruction set

- a. If register r4 has a value 0x f000 000c , what is the value in r0 as the result of running the following ARM assembly language program ?

```
CMP r4, #0
BLE L1
MOV r5, #1
B L2
L1: MOV r5, #2
L2: MOV r0, r5
```

Write the value in decimal: **r0 = 2**

- b. For the following ARM assembly code,
Address code

```
-----
0x0000 1000      Main: MOV r4, #5
0x0000 1004              BL FOO
0x0000 1008              SWI 0x11
0x0000 100C      FOO: MOV r5, #1
0x0000 1010      L1:  CMP r4, #0
0x0000 1014              BLE L2
0x0000 1018              MUL r6, r5, r4
0x0000 101C              MOV r5, r6
0x0000 1020              SUB r4, r4, #1
0x0000 1024              B L1
0x0000 1028      L2:  MOV r0, r5
0x0000 102C              MOV pc, r14
```

- i. When the program halts, what are the values in the following registers?
r0 = 120
r14 = 0x0000 1008
r15 = 0x0000 1008
- ii. How many time has the instruction “MUL r6, r5, r4” been executed?

5
- iii. What does the program compute?

The program computes factorial for the integer stored in r4, in this case, it is $5! = 120$.

4. [30 points] ARM Assembly programming

The following is a C function that takes an integer $n > 0$ and returns $1 + \dots + n$.

```
int sum_to (int n) {  
    if (n<=1) return 1;  
    else  
        return n + sum_to(n-1);  
}
```

- a) You are asked to translate the program into ARM assembly code. You may assume that n is in $r0$, and write the returned value in $r1$.
- b) If $n = 5$, how many activation frames are pushed onto the stack during the execution of the above program.

Answer:

a)

```
sum_to: sub sp, sp, #8  
        str lr, [sp,#0]  
        str r0, [sp,#4]  
        cmp r0,#1  
        bgt else  
        mov r1, #1  
        add sp, sp, #8  
        mov pc, lr  
else: sub r0, r0, #1  
        BL sum_to  
        mov r2, r1  
        ldr r1, [sp, #4]  
        ldr lr, [sp, #0]  
        add sp, sp, #8  
        add r1, r2, r1  
        mov pc, lr
```

b) 5

CISC 260 Machine Organization and Assembly Language

Assignment # 1 Solution

1. [25 points] Convert the following numbers to other data representations. The binary is 8-bit, interpreted as two's complement.

Decimal	Binary	Hexadecimal
-98	1001 1110	0x9E
-39	1101 1001	D9
62	0011 1110	3E
91	0101 1011	0x 5B
-8	1111 1000	0xF8

2. [25 points] ASCII code.

- a) Decode the following bit sequence (expressed in hexadecimal):

0X49034353: I♥CS

- b) Encode the following word to bit sequence (expressed in hexadecimal):

UDCIS: 55 44 43 49 53

3. With $x = 1001\ 0001_{\text{two}}$ and $y = 0100\ 1010_{\text{two}}$ representing two's complement signed integers, perform the following operations, showing all the work:

Answer:

Decimal value of x and y : $x_{10} = -111$ and $y_{10} = 74$

- a. $x + y$:

Expected answer:

$$(-111) + 74 = -37_{10}$$

Actual Answer:

$$\begin{array}{r} 1001\ 0001 \\ +\ 0100\ 1010 \\ \hline 1101\ 1011_2\ (-37_{10}) \end{array}$$

Overflow: No

- b. $x - y$:

Expected answer:

$$(-111) - 74 = -185$$

Actual Answer:

$$y_{10} = 74; y = 0100\ 1010_{\text{two}}$$

$$1\text{'s complement of } y = 1011\ 0101$$

$$2\text{'s complement of } y =$$

$$\begin{array}{r} 1011\ 0101 \\ +\ 1 \\ \hline 1011\ 0110 \end{array}$$

$x - y = x + 2\text{'s complement of } y$:

$$\begin{array}{r} 1001\ 0001 \\ +\ 1011\ 0110 \\ \hline 10100\ 0111 \end{array}$$

Overflow: yes

CISC 260 Machine Organization and Assembly Language

Assignment # 1 Solution

4. [30 points] Write a C program to implement the Booth algorithm for multiplication of signed integers, as discussed in class. You may assume the input a and b are small enough, i.e., only require 16-bit, so that the product $c = a \times b$ can fit into 32-bit. The following is a template for reading two integers a and b, and printing the product $c = a \times b$.

Answer:

Booth's Algorithm:

```
#include <stdio.h>
int multBooth (int q, int m) {
    int a = 0;
    int q_neg1 = 0;
    int q_0 = 0;
    int i = 16;
    while (i > 0) {
        q_0 = q & 1;
        if (q_0 == 1 && q_neg1 == 0) // 10 {
            a -= m;
        }
        else if (q_0 == 0 && q_neg1 == 1) // 01 {
            a += m;
        }
        q_neg1 = q_0;
        m <<= 1;
        q >>= 1;
        i -= 1;
    } return a;
}

void main () {
    int q, m, a;
    printf ("Enter an integer:\n");
    scanf ("%d", &q);
    printf ("Enter an integer:\n");
    scanf ("%d", &m);
    // the code of your subroutine multBooth is called below
    a = multBooth (q, m);
    printf ("the product = %d\n", a);
}
```

CISC 260 Machine Organization and Assembly Language

Assignment # 2 Solution; Fall 2021

1. [25 points] Given the following truth table, where X, Y, and Z are input and W is output, write the canonical expression and generate gate-level logical circuit (draw the wire diagram).

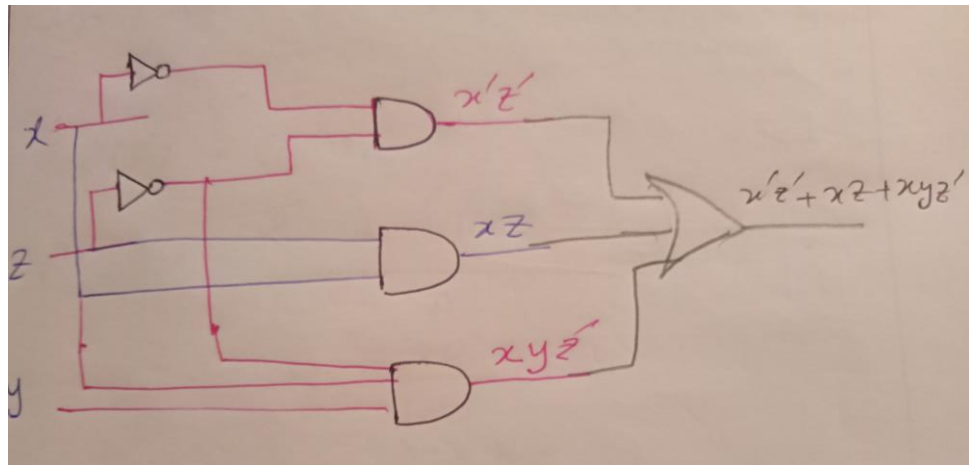
Canonical expression:

$$W = x'y'z' + x'yz' + xy'z + xyz' + xyz$$

$$= x'z'(y' + y) + xz(y' + y) + xyz'$$

$$= x'z' + xz + xyz'$$

- *Multiple Solution possible



2. [25 points] Fill out the truth table for the following circuit. Note that x, y, and z are input, and F is output.

Detailed explanation:

Question - 2 Detailed Explanation

x	y	z	z'	xy	yoz'	x+xz	(xy + yoz')	F =
0	0	0	1	0	0	1	1	1
0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	0	1	1	1
1	0	0	1	0	0	1	1	1
1	0	1	0	0	1	1	0	0
1	1	0	1	1	1	1	0	0
1	1	1	0	1	0	1	0	0

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

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

Final output:

CISC 260 Machine Organization and Assembly Language
Assignment # 2 Solution; Fall 2021

x	y	z	F	Marks
0	0	0	1	4
0	0	1	0	3
0	1	0	0	3
0	1	1	1	3
1	0	0	1	3
1	0	1	0	3
1	1	0	0	3
1	1	1	0	3

3. [25 points] Simply the following Boolean expressions as much as you can.

a) $W = y(xz' + x'z) + y'(xz' + x'z)$

b) $W = xy + xyz + xy'z + x'y'z$

Note: x' , y' and z' stand for $\text{not}(x)$, $\text{not}(y)$, and $\text{not}(z)$ respectively.

Answer:

3(a): $W = y(xz' + x'z) + y'(xz' + x'z)$
 $= (xz' + x'z)(y + y')$
 $= (xz' + x'z) * 1 \quad [(y + y') = 1]$
 $= (xz' + x'z)$

[12.5] 6.5+6

3(b) $W = xy + xyz + xy'z + x'y'z$
 $= xy(1+z) + y'z(x+x') \quad [(1+z)=1 \text{ and } (x+x')=1]$
 $= xy + y'z$

[12.5]: 6.5+6

4. [25 points] You are asked to design a circuit to detect if an overflow occurs when subtracting two integers represented in two's complement: $Z = X - Y$. Let S_z , S_x , and S_y be the sign bit for Z , X , and Y respectively, and they are fed as input to the circuit. Let O be the output bit of the circuit, whose value is 1 if an overflow happens, and 0 if otherwise.

a) Build the truth table for O as a Boolean function of S_x , S_y , and S_z .

b) Write the canonical expression (sum-of-product) for the Boolean function defined in the part a.

c) Implement the Boolean expression defined in part b with a circuit by using AND, OR, and NOT gates. Draw the wiring diagram.

4(a) Truth Table: 10

S_x	S_y	S_z	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0

CISC 260 Machine Organization and Assembly Language
Assignment # 2 Solution; Fall 2021

1	1	1	0
---	---	---	----------

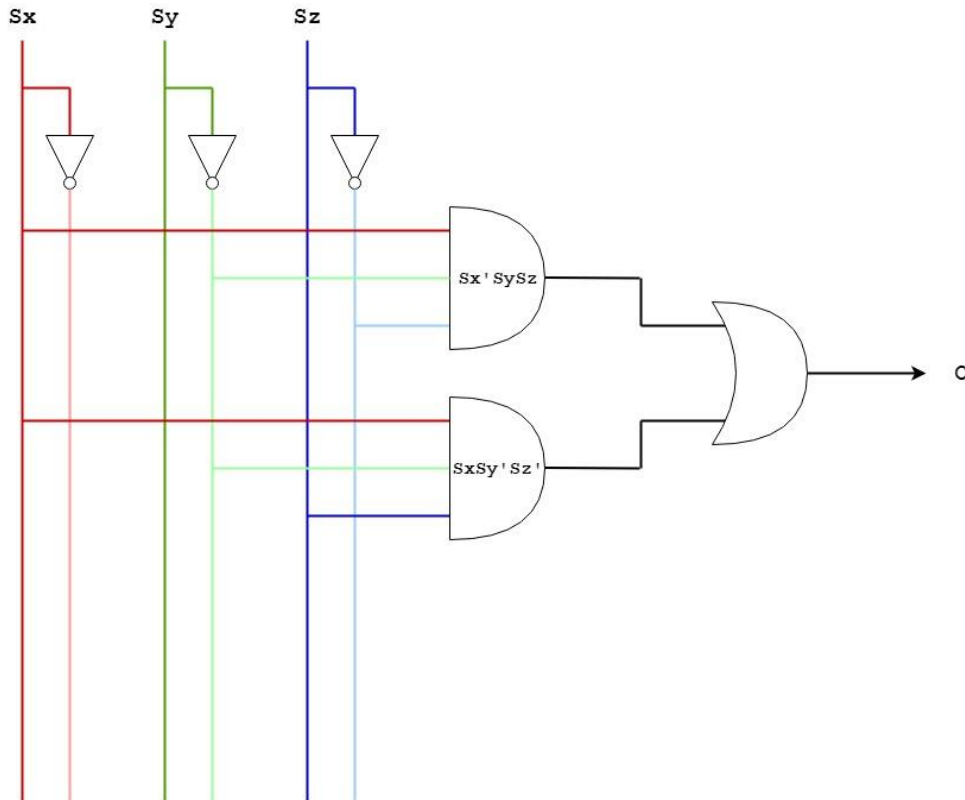
4(b) $O = S'_x S_y S_z + S_x S'_y S'_z$

5

CISC 260 Machine Organization and Assembly Language
Assignment # 2 Solution; Fall 2021

4 (C) Circuit Diagram 15 for wrong equation marks figure 05, for correct equation but little mistake in figure

10



From Zero to One

1

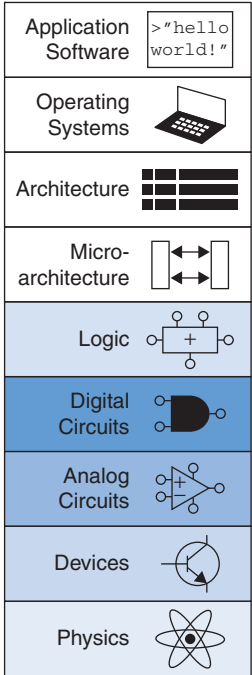
1.1 THE GAME PLAN

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 100 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US \$21 billion in 1985 to \$306 billion in 2013, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and 0's as outputs. We then explore how to combine logic gates into more complicated modules such as adders and memories. Then we shift gears to programming in assembly language, the native tongue of the microprocessor. Finally, we put gates together to build a microprocessor that runs these assembly language programs.

A great advantage of digital systems is that the building blocks are quite simple: just 1's and 0's. They do not require grungy mathematics or a profound knowledge of physics. Instead, the designer's challenge is to combine these simple blocks into complicated systems. A microprocessor may be the first system that you build that is too complex to fit in

- 1.1 The Game Plan
- 1.2 The Art of Managing Complexity
- 1.3 The Digital Abstraction
- 1.4 Number Systems
- 1.5 Logic Gates
- 1.6 Beneath the Digital Abstraction
- 1.7 CMOS Transistors*
- 1.8 Power Consumption*
- 1.9 Summary and a Look Ahead
- Exercises
- Interview Questions



your head all at once. One of the major themes weaved through this book is how to manage complexity.

1.2 THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to managing complexity. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

1.2.1 Abstraction

The critical technique for managing complexity is *abstraction*: hiding details when they are not important. A system can be viewed from many different levels of abstraction. For example, American politicians abstract the world into cities, counties, states, and countries. A county contains multiple cities and a state contains many counties. When a politician is running for president, the politician is mostly interested in how the state as a whole will vote, rather than how each county votes, so the state is the most useful level of abstraction. On the other hand, the Census Bureau measures the population of every city, so the agency must consider the details of a lower level of abstraction.

Figure 1.1 illustrates levels of abstraction for an electronic computer system along with typical building blocks at each level. At the lowest level of abstraction is the physics, the motion of electrons. The behavior of electrons is described by quantum mechanics and Maxwell’s equations. Our system is constructed from electronic *devices* such as transistors (or vacuum tubes, once upon a time). These devices have well-defined connection points called *terminals* and can be modeled by the relationship between voltage and current as measured at each terminal. By abstracting to this device level, we can ignore the individual electrons. The next level of abstraction is *analog circuits*, in which devices are assembled to create components such as amplifiers. Analog circuits input and output a continuous range of voltages. *Digital circuits* such as logic gates restrict the voltages to discrete ranges, which we will use to indicate 0 and 1. In logic design, we build more complex structures, such as adders or memories, from digital circuits.

Microarchitecture links the logic and architecture levels of abstraction. The *architecture* level of abstraction describes a computer from the programmer’s perspective. For example, the Intel x86 architecture used by microprocessors in most *personal computers* (PCs) is defined by a set of

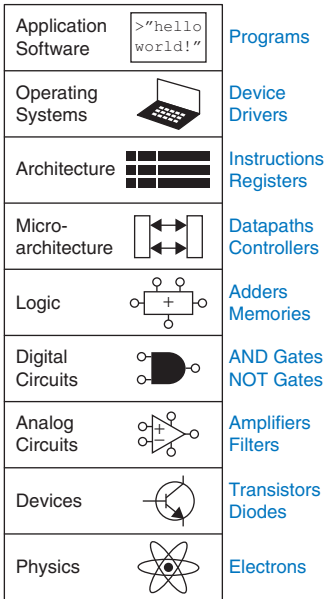


Figure 1.1 Levels of abstraction for an electronic computing system

instructions and registers (memory for temporarily storing variables) that the programmer is allowed to use. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture. A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core i7, the Intel 80486, and the AMD Athlon all implement the x86 architecture with different microarchitectures.

Moving into the software realm, the operating system handles low-level details such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user. Thanks to the power of abstraction, your grandmother can surf the Web without any regard for the quantum vibrations of electrons or the organization of the memory in her computer.

This book focuses on the levels of abstraction from digital circuits through computer architecture. When you are working at one level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working. For example, a computer scientist cannot fully optimize code without understanding the architecture for which the program is being written. A device engineer cannot make wise trade-offs in transistor design without understanding the circuits in which the transistors will be used. We hope that by the time you finish reading this book, you can pick the level of abstraction appropriate to solving your problem and evaluate the impact of your design choices on other levels of abstraction.

Each chapter in this book begins with an abstraction icon indicating the focus of the chapter in deep blue, with secondary topics shown in lighter shades of blue.

1.2.2 Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline. One of the first examples of interchangeable parts was in flintlock rifle manufacturing. Until the early 19th century, rifles were individually crafted by hand. Components purchased from many different craftsmen were carefully filed and fit together by a highly skilled gunmaker. The discipline of interchangeable parts revolutionized the industry. By limiting the components to a standardized set with well-defined tolerances, rifles could be assembled and repaired much faster and with less skill. The gunmaker no longer concerned himself with lower levels of abstraction such as the specific shape of an individual barrel or gunstock.

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a subset of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design. By limiting

ourselves to digital circuits, we can easily combine components into sophisticated systems that ultimately outperform those built from analog components in many applications. For example, digital televisions, compact disks (CDs), and cell phones are replacing their analog predecessors.

1.2.3 The Three-Y's

In addition to abstraction and discipline, designers use the three “-y’s” to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

- ▶ *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- ▶ *Modularity* states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.
- ▶ *Regularity* seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

To illustrate these “-y’s” we return to the example of rifle manufacturing. A flintlock rifle was one of the most intricate objects in common use in the early 19th century. Using the principle of hierarchy, we can break it into components shown in [Figure 1.2](#): the lock, stock, and barrel.

The barrel is the long metal tube through which the bullet is fired. The lock is the firing mechanism. And the stock is the wooden body that holds the parts together and provides a secure grip for the user. In turn, the lock contains the trigger, hammer, flint, frizzen, and pan. Each of these components could be hierarchically described in further detail.

Modularity teaches that each component should have a well-defined function and interface. A function of the stock is to mount the barrel and lock. Its interface consists of its length and the location of its mounting pins. In a modular rifle design, stocks from many different manufacturers can be used with a particular barrel as long as the stock and barrel are of the correct length and have the proper mounting mechanism. A function of the barrel is to impart spin to the bullet so that it travels more accurately. Modularity dictates that there should be no side effects: the design of the stock should not impede the function of the barrel.

Regularity teaches that interchangeable parts are a good idea. With regularity, a damaged barrel can be replaced by an identical part. The barrels can be efficiently built on an assembly line, instead of being painstakingly hand-crafted.

We will return to these principles of hierarchy, modularity, and regularity throughout the book.

Captain Meriwether Lewis of the Lewis and Clark Expedition was one of the early advocates of interchangeable parts for rifles. In 1806, he explained:

The guns of Drewyer and Sergt. Pryor were both out of order. The first was repaired with a new lock, the old one having become unfit for use; the second had the cock screw broken which was replaced by a duplicate which had been prepared for the lock at Harpers Ferry where she was manufactured. But for the precaution taken in bringing on those extra locks, and parts of locks, in addition to the ingenuity of John Shields, most of our guns would at this moment be entirely unfit for use; but fortunately for us I have it in my power here to record that they are all in good order.

See Elliott Coues, ed., *The History of the Lewis and Clark Expedition...* (4 vols), New York: Harper, 1893; reprint, 3 vols, New York: Dover, 3:817.

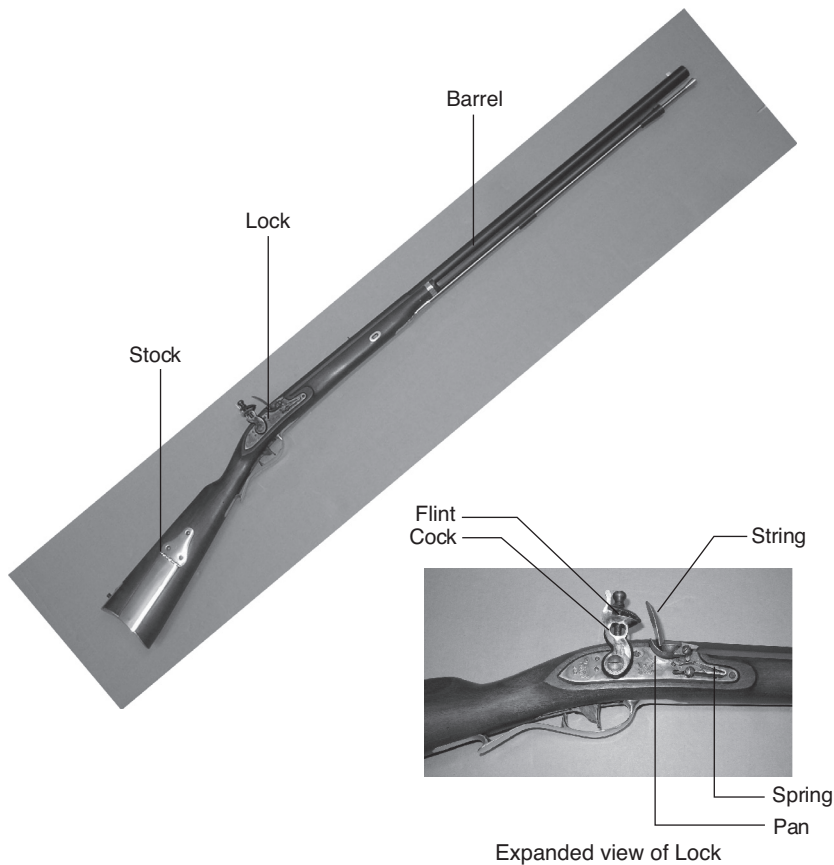
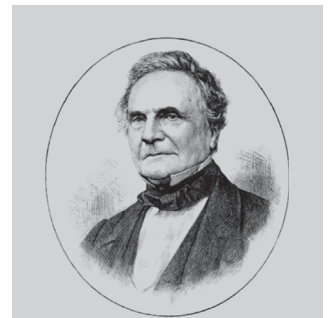


Figure 1.2 Flintlock rifle with a close-up view of the lock

(Image by Euroarms Italia. www.euroarms.net © 2006.)



Charles Babbage, 1791–1871.

Attended Cambridge University and married Georgiana Whitmore in 1814. Invented the Analytical Engine, the world's first mechanical computer. Also invented the cowcatcher and the universal postage rate. Interested in lock-picking, but abhorred street musicians (image courtesy of Fourmilab Switzerland, www.fourmilab.ch).

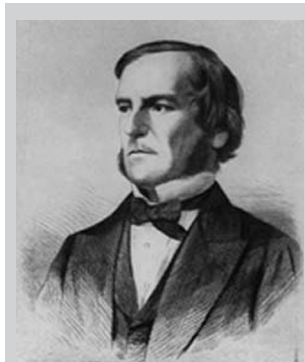
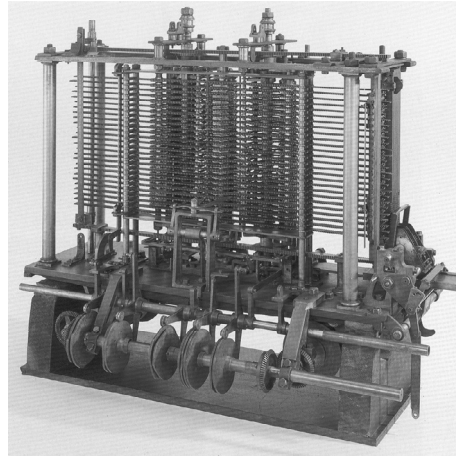
1.3 THE DIGITAL ABSTRACTION

Most physical variables are continuous. For example, the voltage on a wire, the frequency of an oscillation, or the position of a mass are all continuous quantities. Digital systems, on the other hand, represent information with *discrete-valued variables*—that is, variables with a finite number of distinct values.

An early digital system using variables with ten discrete values was Charles Babbage's Analytical Engine. Babbage labored from 1834 to 1871, designing and attempting to build this mechanical computer. The Analytical Engine used gears with ten positions labeled 0 through 9, much like a mechanical odometer in a car. Figure 1.3 shows a prototype of the Analytical Engine, in which each row processes one digit. Babbage chose 25 rows of gears, so the machine has 25-digit precision.

Figure 1.3 Babbage's Analytical Engine, under construction at the time of his death in 1871

(image courtesy of Science Museum/Science and Society Picture Library)



George Boole, 1815–1864. Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote *An Investigation of the Laws of Thought* (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT (image courtesy of the American Institute of Physics).

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a '1' and a low voltage indicates a '0', because it is easier to distinguish between two voltages than ten.

The *amount of information* D in a discrete valued variable with N distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits} \quad (1.1)$$

A binary variable conveys $\log_2 2 = 1$ bit of information. Indeed, the word bit is short for *binary digit*. Each of Babbage's gears carried $\log_2 10 = 3.322$ bits of information because it could be in one of $2^{3.322} = 10$ unique positions. A continuous signal theoretically contains an infinite amount of information because it can take on an infinite number of values. In practice, noise and measurement error limit the information to only 10 to 16 bits for most continuous signals. If the measurement must be made rapidly, the information content is lower (e.g., 8 bits).

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent '1' and zero volts to represent '0'. In this book, we will use the terms '1', TRUE, and HIGH synonymously. Similarly, we will use '0', FALSE, and LOW interchangeably.

The beauty of the *digital abstraction* is that digital designers can focus on 1's and 0's, ignoring whether the Boolean variables are physically represented with specific voltages, rotating gears, or even hydraulic fluid levels. A computer programmer can work without needing to know the intimate

details of the computer hardware. On the other hand, understanding the details of the hardware allows the programmer to optimize the software better for that specific computer.

An individual bit doesn't carry much information. In the next section, we examine how groups of bits can be used to represent numbers. In later chapters, we will also use groups of bits to represent letters and programs.

1.4 NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient. This section introduces the various number systems that will be used throughout the rest of the book.

1.4.1 Decimal Numbers

In elementary school, you learned to count and do arithmetic in *decimal*. Just as you (probably) have ten fingers, there are ten decimal digits: 0, 1, 2, ..., 9. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are 1, 10, 100, 1000, and so on. Decimal numbers are referred to as *base 10*. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.4 shows how the decimal number 9742_{10} is written as the sum of each of its digits multiplied by the weight of the corresponding column.

An N -digit decimal number represents one of 10^N possibilities: 0, 1, 2, 3, ..., $10^N - 1$. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

1.4.2 Binary Numbers

Bits represent one of two values, 0 or 1, and are joined together to form *binary numbers*. Each column of a binary number has twice the weight of the previous column, so binary numbers are *base 2*. In binary, the

$$9742_{10} = 9 \times 10^3 + 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

nine thousands
seven hundreds
four tens
two ones

Figure 1.4 Representation of a decimal number

column weights (again from right to left) are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on. If you work with binary numbers often, you'll save time if you remember these powers of two up to 2^{16} .

An N -bit binary number represents one of 2^N possibilities: 0, 1, 2, 3, ..., $2^N - 1$. Table 1.1 shows 1, 2, 3, and 4-bit binary numbers and their decimal equivalents.

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution: Figure 1.5 shows the conversion.

Table 1.1 Binary numbers and their decimal equivalent

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

1's column

2's column

4's column

8's column

16's column

1

0

1

1

0

$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$

one sixteen

no eight

one four

one two

no one

Figure 1.5 Conversion of a binary number to decimal

Example 1.2 DECIMAL TO BINARY CONVERSION

Convert the decimal number 84₁₀ to binary.

Solution: Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than or equal to the number (in this case, 64). 84 ≥ 64, so there is a 1 in the 64's column, leaving 84 – 64 = 20. 20 < 32, so there is a 0 in the 32's column. 20 ≥ 16, so there is a 1 in the 16's column, leaving 20 – 16 = 4. 4 < 8, so there is a 0 in the 8's column. 4 ≥ 4, so there is a 1 in the 4's column, leaving 4 – 4 = 0. Thus there must be 0's in the 2's and 1's column. Putting this all together, 84₁₀ = 1010100₂.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. 84/2 = 42, so 0 goes in the 1's column. 42/2 = 21, so 0 goes in the 2's column. 21/2 = 10 with a remainder of 1 going in the 4's column. 10/2 = 5, so 0 goes in the 8's column. 5/2 = 2 with a remainder of 1 going in the 16's column. 2/2 = 1, so 0 goes in the 32's column. Finally 1/2 = 0 with a remainder of 1 going in the 64's column. Again, 84₁₀ = 1010100₂.

1.4.3 Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of 2⁴ = 16 possibilities. Hence, it is sometimes more convenient to work in *base 16*, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F, as shown in Table 1.2. Columns in base 16 have weights of 1, 16, 16² (or 256), 16³ (or 4096), and so on.

“Hexadecimal,” a term coined by IBM in 1963, derives from the Greek *hexi* (six) and Latin *decem* (ten). A more proper term would use the Latin *sexa* (six), but *sexadecimal* sounded too risqué.

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

Convert the hexadecimal number 2ED₁₆ to binary and to decimal.

Solution: Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. 2₁₆ = 0010₂, E₁₆ = 1110₂ and D₁₆ = 1101₂, so 2ED₁₆ = 001011101101₂. Conversion to decimal requires the arithmetic shown in Figure 1.6.

Table 1.2 Hexadecimal number system

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Figure 1.6 Conversion of a hexadecimal number to decimal

1's column
16's column
256's column

$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$

two hundred fifty six's fourteen sixteens thirteen ones

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

Solution: Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence $1111010_2 = 7A_{16}$.

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution: Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than or equal to the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving $333 - 256 = 77$. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving $77 - 16 \times 4 = 13$. $13_{10} = D_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14D_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14D_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. $333/16 = 20$ with a remainder of $13_{10} = D_{16}$ going in the 1's column. $20/16 = 1$ with a remainder of 4 going in the 16's column. $1/16 = 0$ with a remainder of 1 going in the 256's column. Again, the result is $14D_{16}$.

1.4.4 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a *byte*. It represents one of $2^8 = 256$ possibilities. The size of objects stored in computer memories is customarily measured in bytes rather than bits.

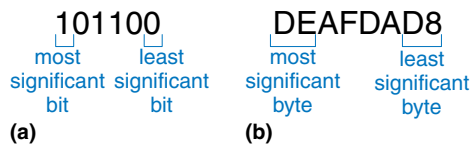
A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities. One hexadecimal digit stores one nibble and two hexadecimal digits store one full byte. Nibbles are no longer a commonly used unit, but the term is cute.

Microprocessors handle data in chunks called *words*. The size of a word depends on the architecture of the microprocessor. When this chapter was written in 2015, most computers had 64-bit processors, indicating that they operate on 64-bit words. At the time, older computers handling 32-bit words were also widely available. Simpler microprocessors, especially those used in gadgets such as toasters, use 8- or 16-bit words.

Within a group of bits, the bit in the 1's column is called the *least significant bit (lsb)*, and the bit at the other end is called the *most significant bit (msb)*, as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a word, the bytes are identified as *least significant byte (LSB)* through *most significant byte (MSB)*, as shown in Figure 1.7(b) for a four-byte number written with eight hexadecimal digits.

A *microprocessor* is a processor built on a single chip. Until the 1970's, processors were too complicated to fit on one chip, so mainframe processors were built from boards containing many chips. Intel introduced the first 4-bit microprocessor, called the 4004, in 1971. Now, even the most sophisticated supercomputers are built using microprocessors. We will use the terms microprocessor and processor interchangeably throughout this book.

Figure 1.7 Least and most significant bits and bytes



By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 2^{10} . For example, 2^{10} bytes is one kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$. If you know $2^{10} \approx 1$ thousand, $2^{20} \approx 1$ million, $2^{30} \approx 1$ billion, and remember the powers of two up to 2^9 , it is easy to estimate any power of two in your head.

Example 1.6 ESTIMATING POWERS OF TWO

Find the approximate value of 2^{24} without using a calculator.

Solution: Split the exponent into a multiple of ten and the remainder.

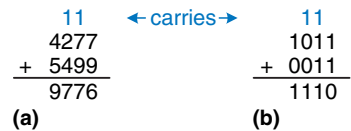
$2^{24} = 2^{20} \times 2^4$. $2^{20} \approx 1$ million. $2^4 = 16$. So $2^{24} \approx 16$ million. Technically, $2^{24} = 16,777,216$, but 16 million is close enough for marketing purposes.

1024 bytes is called a *kilobyte* (KB). 1024 bits is called a *kilobit* (Kb or Kbit). Similarly, MB, Mb, GB, and Gb are used for millions and billions of bytes and bits. Memory capacity is usually measured in bytes. Communication speed is usually measured in bits/sec. For example, the maximum speed of a dial-up modem is usually 56 kbits/sec.

1.4.5 Binary Addition

Binary addition is much like decimal addition, but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column. Figure 1.8 compares addition of decimal and binary numbers. In the right-most column of Figure 1.8(a), $7 + 9 = 16$, which cannot fit in a single digit because it is greater than 9. So we record the 1’s digit, 6, and carry the 10’s digit, 1, over to the next column. Likewise, in binary, if the sum of two numbers is greater than 1, we carry the 2’s digit over to the next column. For example, in the right-most column of Figure 1.8(b),

Figure 1.8 Addition examples showing carries: (a) decimal (b) binary



the sum $1 + 1 = 2_{10} = 10_2$ cannot fit in a single binary digit. So we record the 1's digit (0) and carry the 2's digit (1) of the result to the next column. In the second column, the sum is $1 + 1 + 1 = 3_{10} = 11_2$. Again, we record the 1's digit (1) and carry the 2's digit (1) to the next column. For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*.

Example 1.7 BINARY ADDITION

Compute $0111_2 + 0101_2$.

Solution: Figure 1.9 shows that the sum is 1100_2 . The carries are indicated in blue. We can check our work by repeating the computation in decimal. $0111_2 = 7_{10}$. $0101_2 = 5_{10}$. The sum is $12_{10} = 1100_2$.

$$\begin{array}{r} \text{111} \\ 0111 \\ + 0101 \\ \hline 1100 \end{array}$$

Figure 1.9 Binary addition example

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits. A 4-bit number, for example, has the range $[0, 15]$. 4-bit binary addition overflows if the result exceeds 15. The fifth bit is discarded, producing an incorrect result in the remaining four bits. Overflow can be detected by checking for a carry out of the most significant column.

Example 1.8 ADDITION WITH OVERFLOW

Compute $1101_2 + 0101_2$. Does overflow occur?

Solution: Figure 1.10 shows the sum is 10010_2 . This result overflows the range of a 4-bit binary number. If it must be stored as four bits, the most significant bit is discarded, leaving the incorrect result of 0010_2 . If the computation had been done using numbers with five or more bits, the result 10010_2 would have been correct.

$$\begin{array}{r} \text{111} \\ 1101 \\ + 0101 \\ \hline 10010 \end{array}$$

Figure 1.10 Binary addition example with overflow

1.4.6 Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent *signed* binary numbers; the two most widely employed are called sign/magnitude and two's complement.

Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a minus sign followed by the magnitude. An N -bit sign/magnitude number uses the most significant

The \$7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.



(Photograph courtesy of ESA/CNES/ARIANESPACE-Service Optique CS6.)

bit as the sign and the remaining $N-1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

Example 1.9 SIGN/MAGNITUDE NUMBERS

Write 5 and -5 as 4-bit sign/magnitude numbers

Solution: Both numbers have a magnitude of $5_{10} = 101_2$. Thus, $5_{10} = 0101_2$ and $-5_{10} = 1101_2$.

Unfortunately, ordinary binary addition does not work for sign/magnitude numbers. For example, using ordinary addition on $-5_{10} + 5_{10}$ gives $1101_2 + 0101_2 = 10010_2$, which is nonsense.

An N -bit sign/magnitude number spans the range $[-2^{N-1} + 1, 2^{N-1} - 1]$. Sign/magnitude numbers are slightly odd in that both $+0$ and -0 exist. Both indicate zero. As you may expect, it can be troublesome to have two different representations for the same number.

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1} . They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00...000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01...111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10...000_2 = -2^{N-1}$. And -1 is written as all ones: $11...111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. This is useful to find the representation of a negative number or to determine the magnitude of a negative number.

Example 1.10 TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution: Start with $+2_{10} = 0010_2$. To get -2_{10} , invert the bits and add 1. Inverting 0010_2 produces 1101_2 . $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2 .

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

Find the decimal value of the two's complement number 1001_2 .

Solution: 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N -bit numbers, the carry out of the N th bit (i.e., the $N + 1^{\text{th}}$ result bit) is discarded.

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2 .

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution: (a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011_2$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

The two's complement of 0 is found by inverting all the bits (producing $11\dots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate -0 . Zero is considered positive because its sign bit is 0.

Like unsigned numbers, N -bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: -8 to 7. In general, the range of an N -bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no -0 . The most negative number $10\dots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\dots111_2$) and adding 1, which produces $10\dots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.

Adding two N -bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1} . Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution: $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

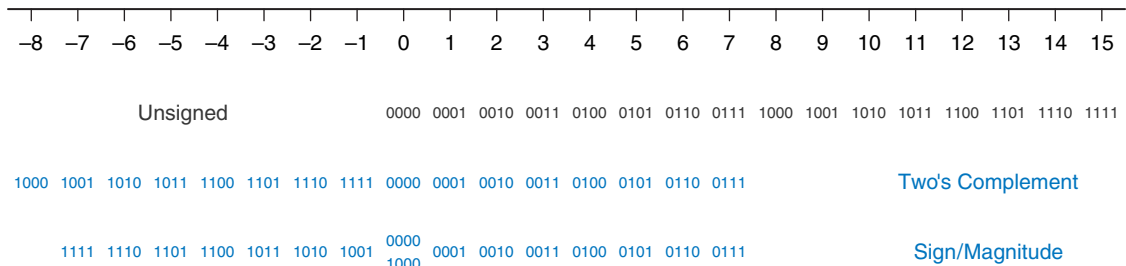
When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and -3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively.

Comparison of Number Systems

The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude. Table 1.3 compares the range of N -bit numbers in each of these three systems. Two's complement numbers are convenient because they represent both positive and negative integers and because ordinary addition works for all numbers. Subtraction is performed by negating the second number (i.e., taking the two's

Table 1.3 Range of N -bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

**Figure 1.11** Number line and 4-bit binary encodings

complement), and then adding. Unless stated otherwise, assume that all signed binary numbers use two's complement representation.

Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range $[0, 15]$ in regular binary order. Two's complement numbers span the range $[-8, 7]$. The nonnegative numbers $[0, 7]$ share the same encodings as unsigned numbers. The negative numbers $[-8, -1]$ are encoded such that a larger unsigned binary value represents a number closer to 0. Notice that the weird number, 1000, represents -8 and has no positive counterpart. Sign/magnitude numbers span the range $[-7, 7]$. The most significant bit is the sign bit. The positive numbers $[1, 7]$ share the same encodings as unsigned numbers. The negative numbers are symmetric but have the sign bit set. 0 is represented by both 0000 and 1000. Thus, N -bit sign/magnitude numbers represent only $2^N - 1$ integers because of the two representations for 0.

1.5 LOGIC GATES

Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input (or inputs) and the output. Inputs are usually drawn on

the left (or top) and outputs on the right (or bottom). Digital designers typically use letters near the beginning of the alphabet for gate inputs and the letter Y for the gate output. The relationship between the inputs and the output can be described with a truth table or a Boolean equation. A *truth table* lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A *Boolean equation* is a mathematical expression using binary variables.

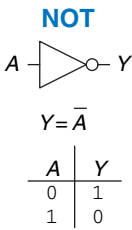


Figure 1.12 NOT gate

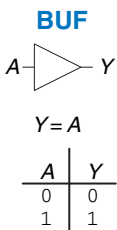


Figure 1.13 Buffer

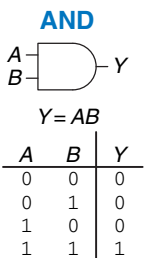


Figure 1.14 AND gate

According to Larry Wall, inventor of the Perl programming language, “the three principal virtues of a programmer are Laziness, Impatience, and Hubris.”

1.5.1 NOT Gate

A *NOT gate* has one input, A, and one output, Y, as shown in Figure 1.12. The NOT gate’s output is the inverse of its input. If A is FALSE, then Y is TRUE. If A is TRUE, then Y is FALSE. This relationship is summarized by the truth table and Boolean equation in the figure. The line over A in the Boolean equation is pronounced NOT, so $Y = \bar{A}$ is read “Y equals NOT A.” The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$ or $Y = \sim A$. We will use $Y = \bar{A}$ exclusively, but don’t be puzzled if you encounter another notation elsewhere.

1.5.2 Buffer

The other one-input logic gate is called a *buffer* and is shown in Figure 1.13. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. However, from the analog point of view, the buffer might have desirable characteristics such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates. This is an example of why we need to consider multiple levels of abstraction to fully understand a system; the digital abstraction hides the real purpose of a buffer.

The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.

1.5.3 AND Gate

Two-input logic gates are more interesting. The *AND gate* shown in Figure 1.14 produces a TRUE output, Y, if and only if both A and B are TRUE. Otherwise, the output is FALSE. By convention, the inputs are listed in the order 00, 01, 10, 11, as if you were counting in binary. The Boolean equation for an AND gate can be written in several ways: $Y = A \cdot B$, $Y = AB$, or $Y = A \cap B$. The \cap symbol is pronounced “intersection” and is preferred by logicians. We prefer $Y = AB$, read “Y equals A and B,” because we are lazy.

1.5.4 OR Gate

The OR gate shown in Figure 1.15 produces a TRUE output, Y, if either A or B (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$. The \cup symbol is pronounced union and is preferred by logicians. Digital designers normally use the + notation, $Y = A + B$ is pronounced “Y equals A or B.”

1.5.5 Other Two-Input Gates

Figure 1.16 shows other common two-input logic gates. XOR (exclusive OR, pronounced “ex-OR”) is TRUE if A or B, but not both, are TRUE. The XOR operation is indicated by \oplus , a plus sign with a circle around it. Any gate can be followed by a bubble to invert its operation. The NAND gate performs NOT AND. Its output is TRUE unless both inputs are TRUE. The NOR gate performs NOT OR. Its output is TRUE if neither A nor B is TRUE. An N-input XOR gate is sometimes called a *parity* gate and produces a TRUE output if an odd number of inputs are TRUE. As with two-input gates, the input combinations in the truth table are listed in counting order.


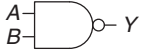
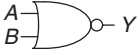
XOR			NAND			NOR		
								
$Y = A \oplus B$			$Y = \overline{AB}$			$Y = \overline{A + B}$		
A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	1
0	1	1	0	1	1	0	1	0
1	0	1	1	0	1	1	0	0
1	1	0	1	1	0	1	1	0

Figure 1.16 More two-input logic gates

Example 1.15 XNOR GATE

Figure 1.17 shows the symbol and Boolean equation for a two-input XNOR gate that performs the inverse of an XOR. Complete the truth table.

Solution: Figure 1.18 shows the truth table. The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an *equality* gate because its output is TRUE when the inputs are equal.

1.5.6 Multiple-Input Gates

Many Boolean functions of three or more inputs exist. The most common are AND, OR, XOR, NAND, NOR, and XNOR. An N-input AND gate

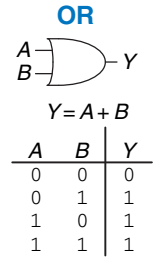


Figure 1.15 OR gate

A silly way to remember the OR symbol is that its input side is curved like Pacman's mouth, so the gate is hungry and willing to eat any TRUE inputs it can find!

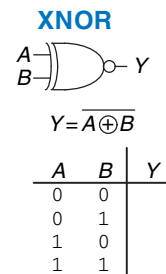
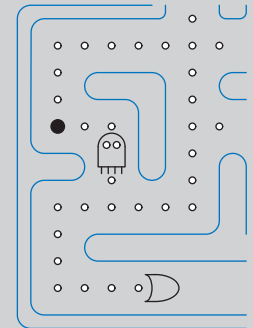


Figure 1.17 XNOR gate

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

Figure 1.18 XNOR truth table

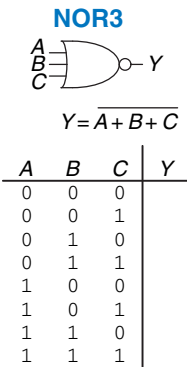


Figure 1.19 Three-input NOR gate

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

Figure 1.20 Three-input NOR truth table

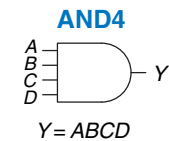


Figure 1.21 Four-input AND gate

produces a TRUE output when all N inputs are TRUE. An N -input OR gate produces a TRUE output when at least one input is TRUE.

Example 1.16 THREE-INPUT NOR GATE

Figure 1.19 shows the symbol and Boolean equation for a three-input NOR gate. Complete the truth table.

Solution: Figure 1.20 shows the truth table. The output is TRUE only if none of the inputs are TRUE.

Example 1.17 FOUR-INPUT AND GATE

Figure 1.21 shows the symbol and Boolean equation for a four-input AND gate. Create a truth table.

Solution: Figure 1.22 shows the truth table. The output is TRUE only if all of the inputs are TRUE.

1.6 BENEATH THE DIGITAL ABSTRACTION

A digital system uses discrete-valued variables. However, the variables are represented by continuous physical quantities such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder. Hence, the designer must choose a way to relate the continuous value to the discrete value.

For example, consider representing a binary signal A with a voltage on a wire. Let 0 volts (V) indicate $A = 0$ and 5 V indicate $A = 1$. Any real system must tolerate some noise, so 4.97 V probably ought to be interpreted as $A = 1$ as well. But what about 4.3 V? Or 2.8 V? Or 2.500000 V?

1.6.1 Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called V_{DD} . In 1970's and 1980's technology, V_{DD} was generally 5 V. As chips have progressed to smaller transistors, V_{DD} has dropped to 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, or even lower to save power and avoid overloading the transistors.

1.6.2 Logic Levels

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown in Figure 1.23. The first gate is called the *driver* and the second gate is called the *receiver*. The output of the driver is

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

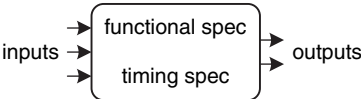


Figure 2.1 Circuit as a black box with inputs, outputs, and specifications

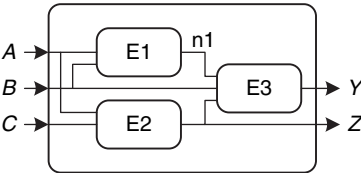


Figure 2.2 Elements and nodes

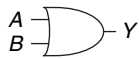
2.1	Introduction
2.2	Boolean Equations
2.3	Boolean Algebra
2.4	From Logic to Gates
2.5	Multilevel Combinational Logic
2.6	X's and Z's, Oh My
2.7	Karnaugh Maps
2.8	Combinational Building Blocks
2.9	Timing
2.10	Summary
	Exercises
	Interview Questions

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

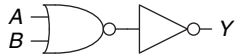


$$Y = F(A, B) = A + B$$

Figure 2.3 Combinational logic circuit

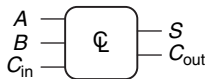


(a)



(b)

Figure 2.4 Two OR implementations



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 2.5 Multiple-output combinational circuit



(a)



(b)

Figure 2.6 Slash notation for multiple signals

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.

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 \mathcal{C} 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 $Y = A \text{ OR } B$.

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.

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} .

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.

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.

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.

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 application of abstraction and modularity. Building the circuit out of

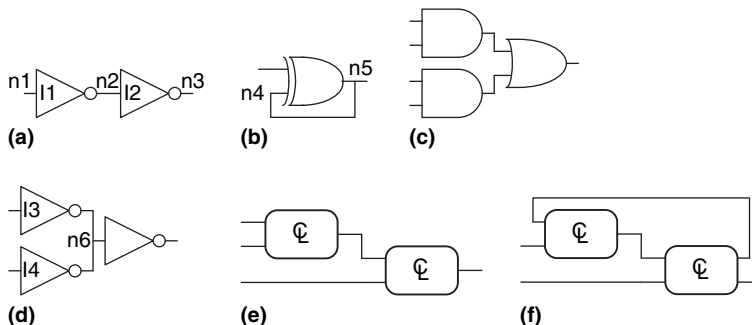


Figure 2.7 Example circuits

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.

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

(2.1)

A	B	Y	minterm	minterm name
0	0	0	$\overline{A} \overline{B}$	m_0
0	1	1	$\overline{A} B$	m_1
1	0	0	$A \overline{B}$	m_2
1	1	0	$A B$	m_3

Figure 2.8 Truth table and minterms

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 $A = 0$, $B = 0$. The minterms are

numbered starting with 0; the top row corresponds to minterm 0, m_0 , the next row to minterm 1, m_1 , and so on.

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.

The sum-of-products canonical form can also be written in *sigma notation* using the summation symbol, Σ . With this notation, the function from Figure 2.9 would be written as:

$$F(A, B) = \Sigma(m_1, m_3) \quad (2.2)$$

or

$$F(A, B) = \Sigma(1, 3)$$

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: $E = \overline{A}\overline{R}$ or $E = \Sigma(0)$. 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: $E = A \text{ NOR } R = \overline{A + R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}\overline{R}$ 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

$$Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C \quad (2.3)$$

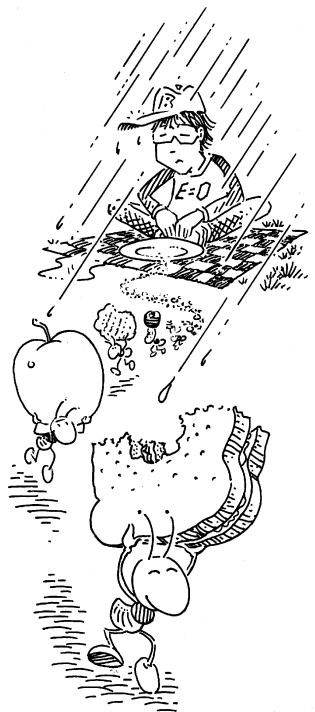
or

$$Y = \Sigma(0, 4, 5)$$

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\overline{B}$	m_0
0	1	1	$\overline{A}B$	m_1
1	0	0	$A\overline{B}$	m_2
1	1	1	AB	m_3

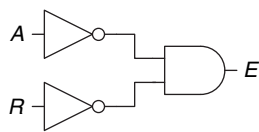
Figure 2.9 Truth table with multiple TRUE minterms

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



A	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	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \bar{B}$	M_1
1	0	0	$\bar{A} + B$	M_2
1	1	1	$\bar{A} + \bar{B}$	M_3

Figure 2.13 Truth table with multiple FALSE maxterms

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.

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. The product-of-sums canonical form can also be written in *pi notation* using the product symbol, Π .

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 $Y = (A + B)(\bar{A} + \bar{B})$ or, using pi notation, $Y = \Pi(M_0, M_2)$ or $Y = \Pi(0, 2)$. 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, $(\bar{A} + \bar{B})$, 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 $E = (A + \bar{R})(\bar{A} + R)(\bar{A} + \bar{R})$ or $E = \Pi(1, 2, 3)$. This is uglier than the sum-of-products equation, $E = \bar{A}\bar{R}$, but the two equations are logically equivalent.

Sum-of-products produces a shorter equation 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.

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

These theorems have great practical significance, because they teach us how to simplify logic to produce smaller and less costly circuits.

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators \bullet (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 , $B \text{ AND } 1 = B$. Its dual states that $B \text{ OR } 0 = B$. 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 $B \text{ AND } 0$ 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 $B \text{ OR } 1$ is always equal to 1. Hence, 1 is the null element for the OR operation. In hardware,

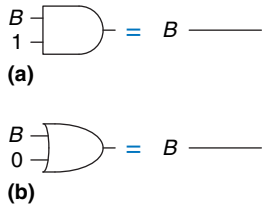


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1'

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.)

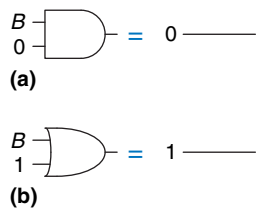


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2'

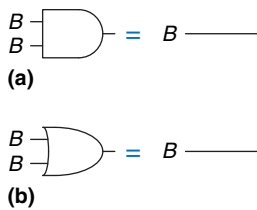


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3'

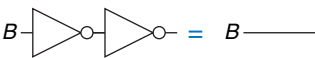


Figure 2.17 Involution theorem in hardware: T4

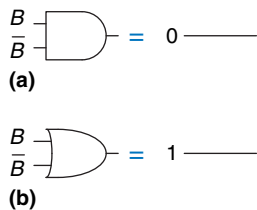


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5'

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

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 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	$\overline{B_0 \bullet B_1 \bullet B_2 \dots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \dots)$	T12'	$\overline{B_0 + B_1 + B_2 \dots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \dots)$	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.

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

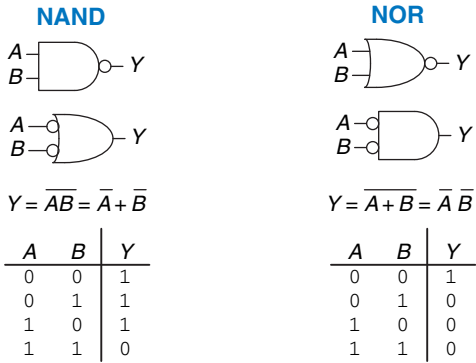



Figure 2.19 De Morgan equivalent gates



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 ."

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	\bar{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 \bar{Y}

A	B	Y	\bar{Y}	minterm
0	0	0	1	$\bar{A}\bar{B}$
0	1	0	1	$\bar{A}B$
1	0	1	0	$A\bar{B}$
1	1	1	0	AB

Figure 2.21 Truth table showing minterms for \bar{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 \bar{Y} . Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of \bar{Y} .

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

$$\bar{Y} = \bar{A}\bar{B} + \bar{A}B \quad (2.4)$$

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

$$\bar{\bar{Y}} = Y = \overline{\bar{A}\bar{B} + \bar{A}B} = (\overline{\bar{A}\bar{B}})(\overline{\bar{A}B}) = (A + B)(A + \bar{B}) \quad (2.5)$$

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 USING PERFECT INDUCTION

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 $BC + \bar{B}D + CD = BC + \bar{B}D$ for all cases, the theorem is proved.

B	C	D	$BC + \bar{B}D + CD$	$BC + \bar{B}D$
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: $Y = \bar{A}B + AB$. By Theorem T10, the equation simplifies to $Y = B$. 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 $PA + P\bar{A} = P$, 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 in the equation 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 2.3: $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$.

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 $\bar{A}\bar{B}\bar{C}$ and $A\bar{B}\bar{C}$ differ only in the variable A . So we combined the minterms to form $\bar{B}\bar{C}$. However, if we look at the original equation, we note that the last two minterms $A\bar{B}\bar{C}$ and $A\bar{B}C$ also differ by a single literal (C and \bar{C}). Thus, using the same method, we could have combined these two minterms to form the minterm $A\bar{B}$. We say that implicants $\bar{B}\bar{C}$ and $A\bar{B}$ *share* the minterm $A\bar{B}\bar{C}$.

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: $B = B + B + B + B \dots$. Using this principle, we simplify the equation completely to its two prime implicants, $\bar{B}\bar{C} + A\bar{B}$, as shown in Table 2.5.

Table 2.4 Equation minimization

Step	Equation	Justification
	$\overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} C$	
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
	$\overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} C$	
1	$\overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} C$	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 2.3:

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

The labs that accompany this textbook (see Preface) show how to use *computer-aided design* (CAD) tools to design, simulate, and test digital circuits.

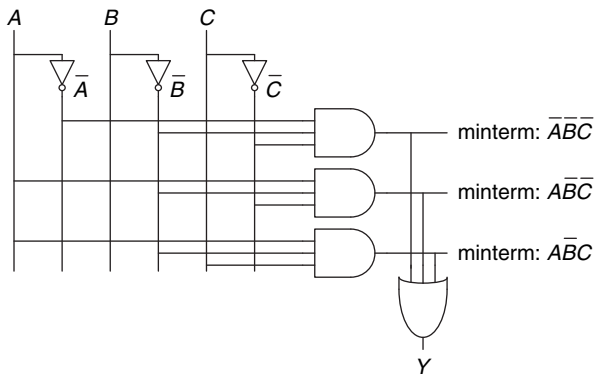


Figure 2.23 Schematic of $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + ABC$

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.

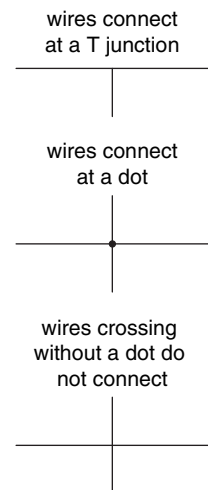


Figure 2.24 Wire connections

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 $\bar{B}\bar{C}$ is an

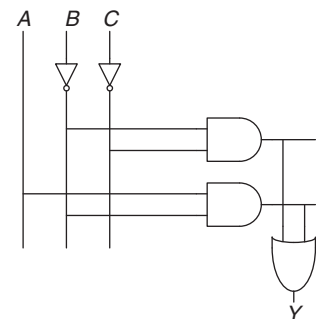


Figure 2.25 Schematic of $Y = \bar{B}\bar{C} + A\bar{B}$

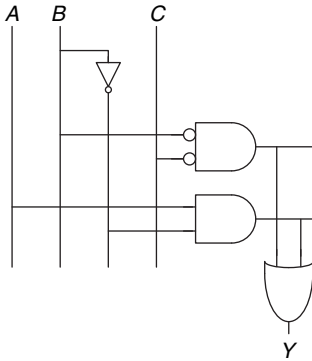


Figure 2.26 Schematic using fewer gates

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 BTB¹ party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, A_3, \dots, A_0 , and four outputs, Y_3, \dots, 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 $Y_2 = \bar{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher priority inputs is asserted: $Y_1 = \bar{A}_3 \bar{A}_2 A_1$. And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \bar{A}_3 \bar{A}_2 \bar{A}_1 A_0$. 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.

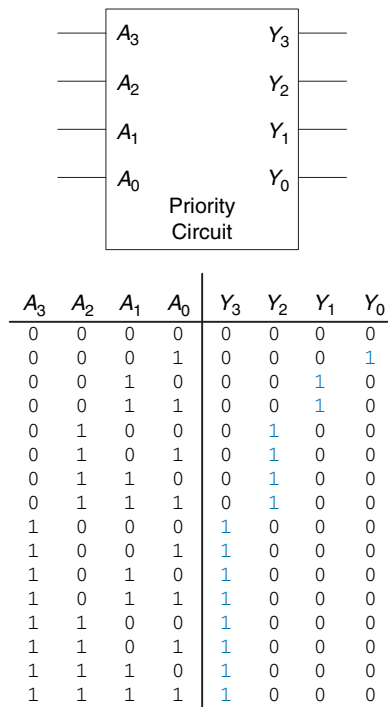


Figure 2.27 Priority circuit

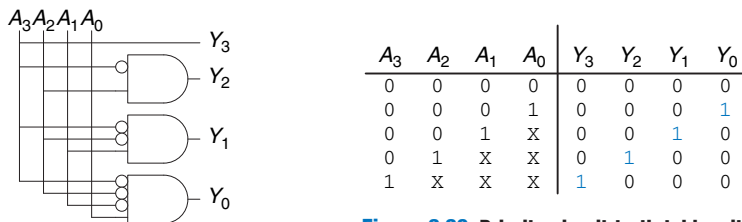


Figure 2.28 Priority circuit schematic

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

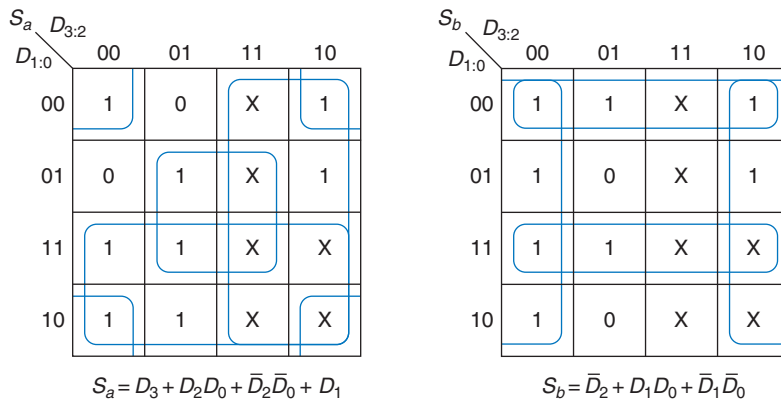


Figure 2.53 K-map solution with don't cares

human with a bit of experience can find a good solution by inspection. Neither of the authors has ever used a Karnaugh map in real life to solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.

2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs D_0 and D_1 , a select input S , and one output Y . The multiplexer chooses between the two data inputs based on the select: if $S=0$, $Y=D_0$, and if $S=1$, $Y=D_1$. S is also called a *control signal* because it controls what the multiplexer does.

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 2:1 multiplexer symbol and truth table

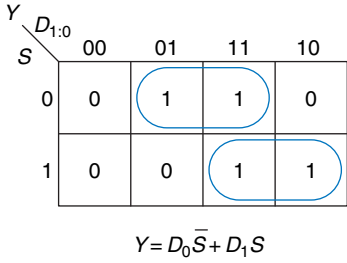


Figure 2.55 2:1 multiplexer implementation using two-level logic

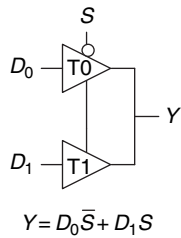
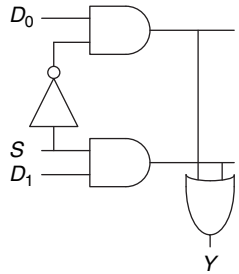


Figure 2.56 Multiplexer using tristate buffers

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

with a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

Alternatively, multiplexers can be built from tristate buffers as shown in Figure 2.56. The tristate enables are arranged such that, at all times, exactly one tristate buffer is active. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y . When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y .

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N :1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

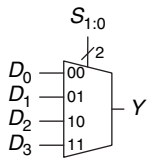


Figure 2.57 4:1 multiplexer

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

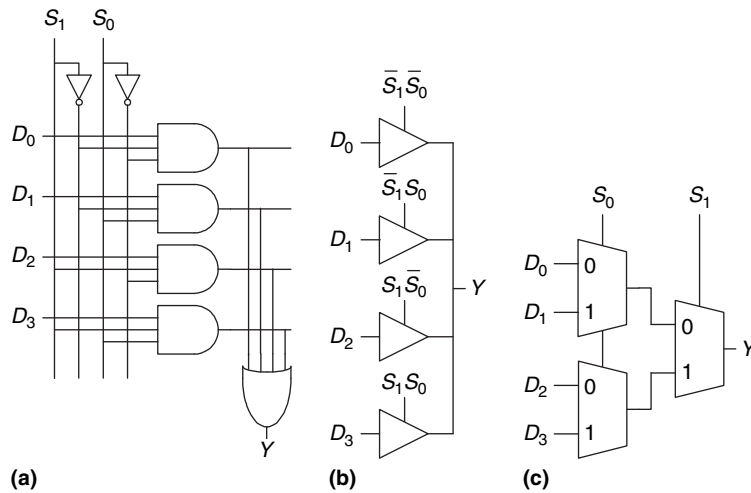


Figure 2.58 4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

AND gate. The inputs, A and B , serve as select lines. The multiplexer data inputs are connected to 0 or 1 according to the corresponding row of the truth table. In general, a 2^N -input multiplexer can be programmed to perform any N -input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1} -input multiplexer to perform any N -input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table, and then combine pairs of rows to eliminate the right-most input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of B . When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.

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

$Y = AB$

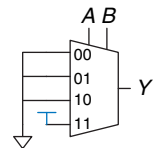


Figure 2.59 4:1 multiplexer implementation of two-input AND function

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = AB\bar{B} + \bar{B}\bar{C} + \bar{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution: Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table where each row in the truth table corresponds to a multiplexer input.

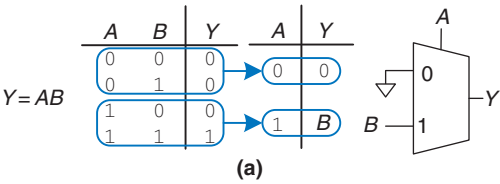


Figure 2.60 Multiplexer logic using variable inputs

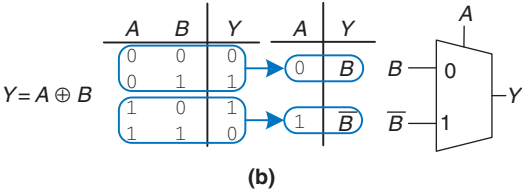
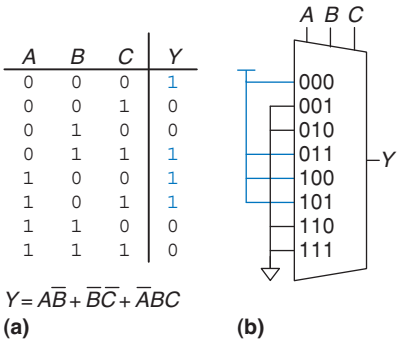


Figure 2.61 Alyssa's circuit:
(a) truth table, (b) 8:1 multiplexer implementation



Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidentally powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution: Alyssa reduces her truth table to four rows by letting the output depend on C . (She could also have chosen to rearrange the columns of the truth table to let the output depend on A or B .) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is “hot” (HIGH) at a given time.

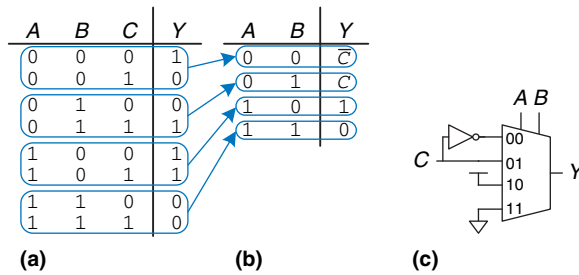
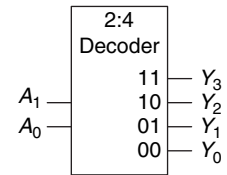


Figure 2.62 Alyssa's new circuit

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND, OR, and NOT gates.

Solution: Figure 2.64 shows an implementation for the 2:4 decoder using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an $N:2^N$ decoder can be constructed from 2^N N -input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A}_1\overline{A}_0$. This fact will be handy when using decoders with other digital building blocks.



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

Figure 2.63 2:4 decoder

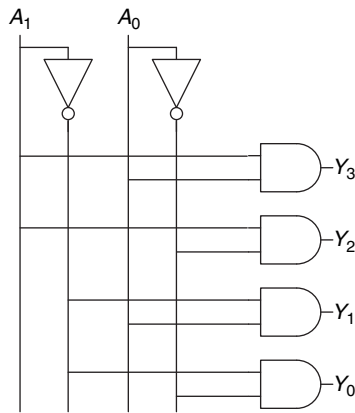


Figure 2.64 2:4 decoder implementation

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all the minterms in the function. In Figure 2.65, $Y = \overline{A}\overline{B} + AB = \overline{A \oplus B}$.

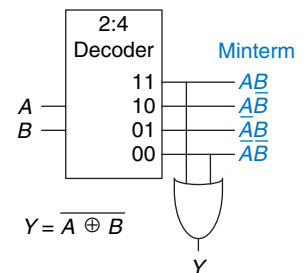


Figure 2.65 Logic function using decoder

Sequential Logic Design

3

3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase speed.

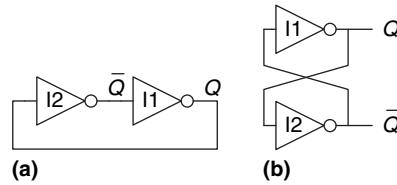
3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs, Q and \overline{Q} .

3.1	Introduction
3.2	Latches and Flip-Flops
3.3	Synchronous Logic Design
3.4	Finite State Machines
3.5	Timing of Sequential Logic
3.6	Parallelism
3.7	Summary
	Exercises
	Interview Questions

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Figure 3.1 Cross-coupled inverter pair



Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on \overline{Q} , and \overline{Q} depends on Q .

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- ▶ **Case I: $Q = 0$**
As shown in Figure 3.2(a), I2 receives a FALSE input, Q , so it produces a TRUE output on \overline{Q} . I1 receives a TRUE input, \overline{Q} , so it produces a FALSE output on Q . This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.
- ▶ **Case II: $Q = 1$**
As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on \overline{Q} . I1 receives a FALSE input and produces a TRUE output on Q . This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be *bistable*. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q . The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, \overline{Q} , but \overline{Q} does not contain any additional information because if Q is known, \overline{Q} is also known. On the other hand, \overline{Q} is also an acceptable choice for the state variable.

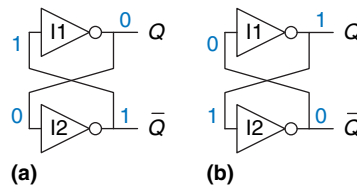


Figure 3.2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R , and two outputs, Q and \bar{Q} . The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q .

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S .

- ▶ *Case I:* $R = 1, S = 0$
N1 sees at least one TRUE input, R , so it produces a FALSE output on Q . N2 sees both Q and S FALSE, so it produces a TRUE output on \bar{Q} .
- ▶ *Case II:* $R = 0, S = 1$
N1 receives inputs of 0 and \bar{Q} . Because we don't yet know \bar{Q} , we can't determine the output Q . N2 receives at least one TRUE input, S , so it produces a FALSE output on \bar{Q} . Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.
- ▶ *Case III:* $R = 1, S = 1$
N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and \bar{Q} are both FALSE.
- ▶ *Case IV:* $R = 0, S = 0$
N1 receives inputs of 0 and \bar{Q} . Because we don't yet know \bar{Q} , we can't determine the output. N2 receives inputs of 0 and Q . Because we don't yet know Q , we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these subcases.

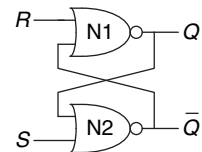
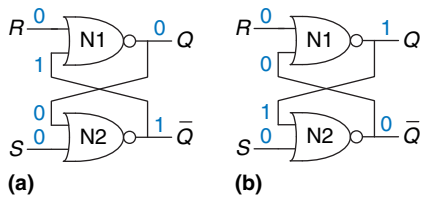


Figure 3.3 SR latch schematic

Figure 3.4 Bistable states of SR latch



- ▶ *Case IVa: $Q = 0$*
Because S and Q are FALSE, N2 produces a TRUE output on \bar{Q} , as shown in Figure 3.4(a). Now N1 receives one TRUE input, \bar{Q} , so its output, Q , is FALSE, just as we had assumed.
- ▶ *Case IVb: $Q = 1$*
Because Q is TRUE, N2 produces a FALSE output on \bar{Q} , as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and \bar{Q} , so its output, Q , is TRUE, just as we had assumed.

Putting this all together, suppose Q has some known prior value, which we will call Q_{prev} , before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev} , and \bar{Q} will be its complement, \bar{Q}_{prev} . This circuit has memory.

Case	S	R	Q	\bar{Q}
IV	0	0	Q_{prev}	\bar{Q}_{prev}
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 SR latch truth table

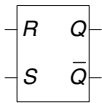


Figure 3.6 SR latch symbol

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and \bar{Q} , are normally complementary. When R is asserted, Q is reset to 0 and \bar{Q} does the opposite. When S is asserted, Q is set to 1 and \bar{Q} does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev} . Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q . However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be

accounted for by the single state variable Q . No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D , controls what the next state should be. The *clock* input, CLK , controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes \bar{D} , S , and R . If $CLK = 0$, both S and R are FALSE, regardless of the value of D . If $CLK = 1$, one AND gate will produce TRUE and the other FALSE, depending on the value of D . Given S and R , Q and \bar{Q} are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev} . When $CLK = 1$, $Q = D$. In all cases, \bar{Q} is the complement of Q , as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q , and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

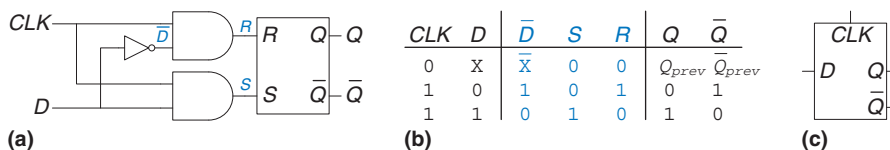


Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

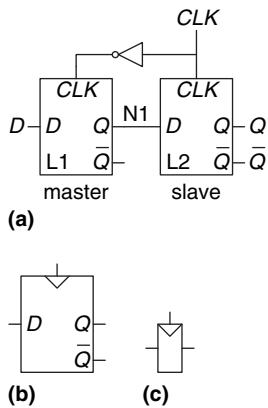


Figure 3.8 D flip-flop:
(a) schematic, (b) symbol,
(c) condensed symbol

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term flip-flop or latch by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

3.2.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the \overline{Q} output is not needed, the symbol is often condensed as in Figure 3.8(c).

When $CLK = 0$, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at N1 propagates through to Q , but N1 is cut off from D . Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q .

In other words, a *D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The \overline{Q} output is often omitted when it is not needed.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution: A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation using transmission gates.

3.2.4 Register

An N -bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time. Registers are the key building block of most sequential circuits. Figure 3.9

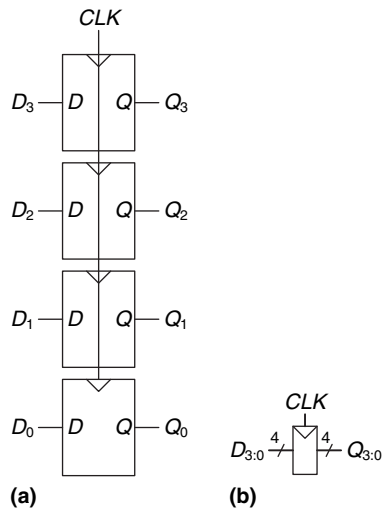


Figure 3.9 A 4-bit register:
(a) schematic and (b) symbol

shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer chooses whether to pass the value at *D*, if *EN* is TRUE, or to recycle the old state from *Q*, if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*. If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is

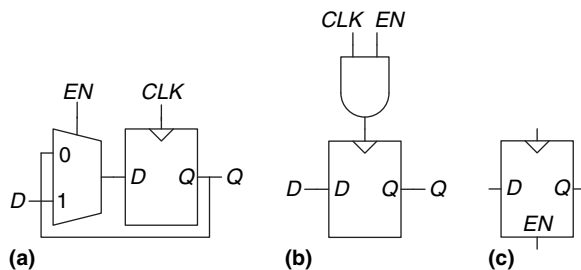


Figure 3.10 Enabled flip-flop:
(a, b) schematics, (c) symbol

Digital Building Blocks

5

5.1 INTRODUCTION

Up to this point, we have examined the design of combinational and sequential circuits using Boolean equations, schematics, and HDLs. This chapter introduces more elaborate combinational and sequential building blocks used in digital systems. These blocks include arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are not only useful in their own right, but they also demonstrate the principles of hierarchy, modularity, and regularity. The building blocks are hierarchically assembled from simpler components such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes. In Chapter 7, we use many of these building blocks to build a microprocessor.

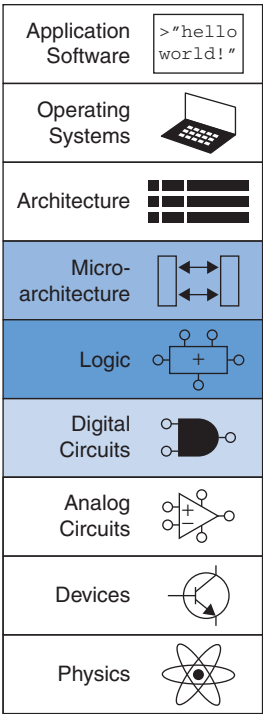
5.2 ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

5.2.1 Addition

Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to N -bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

- 5.1 [Introduction](#)
- 5.2 [Arithmetic Circuits](#)
- 5.3 [Number Systems](#)
- 5.4 [Sequential Building Blocks](#)
- 5.5 [Memory Arrays](#)
- 5.6 [Logic Arrays](#)
- 5.7 [Summary](#)
- [Exercises](#)
- [Interview Questions](#)



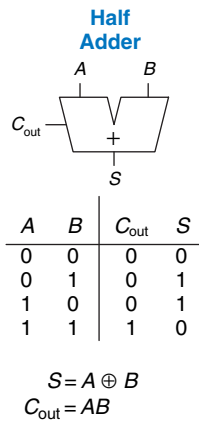


Figure 5.1 1-bit half adder

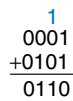


Figure 5.2 Carry bit

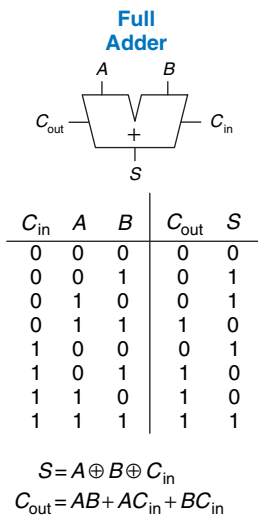


Figure 5.3 1-bit full adder

Half Adder

We begin by building a 1-bit *half adder*. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half adder can be built from an XOR gate and an AND gate.

In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output C_{out} of the first column of 1-bit addition and the input C_{in} to the second column of addition. However, the half adder lacks a C_{in} input to accept C_{out} of the previous column. The *full adder*, described in the next section, solves this problem.

Full Adder

A *full adder*, introduced in Section 2.1, accepts the carry in C_{in} as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in C_{in} to produce an N-bit result S and a carry out C_{out}. It is commonly called a *carry propagate adder* (CPA) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and prefix adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. It is a good application of modularity and regularity: the full adder module is reused many times to form a larger system. The ripple-carry adder has the disadvantage of being slow when N is large. S₃₁ depends on C₃₀, which depends on C₂₉, which depends on C₂₈, and so forth all the way back to C_{in}, as shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder, t_{ripple},

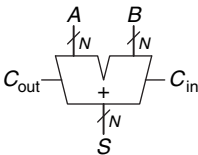


Figure 5.4 Carry propagate adder

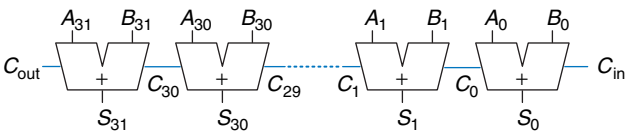


Figure 5.5 32-bit ripple-carry adder

grows directly with the number of bits, as given in Equation 5.1, where t_{FA} is the delay of a full adder.

$$t_{\text{ripple}} = Nt_{FA} \quad (5.1)$$

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder (CLA) is another type of carry propagate adder that solves this problem by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to *look ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder may be divided into eight 4-bit blocks.

CLAs use *generate* (G) and *propagate* (P) signals that describe how a column or block determines the carry out. The i th column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The i th column of an adder is guaranteed to generate a carry C_i if A_i and B_i are both 1. Hence G_i , the generate signal for column i , is calculated as $G_i = A_i B_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The i th column will propagate a carry in, C_{i-1} , if either A_i or B_i is 1. Thus, $P_i = A_i + B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the adder. The i th column of an adder will generate a carry out C_i if it either generates a carry, G_i , or propagates a carry in, $P_i C_{i-1}$. In equation form,

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1} \quad (5.2)$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j .

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1 G_0)) \quad (5.3)$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3 P_2 P_1 P_0 \quad (5.4)$$

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i , using the carry in to the block, C_{j-1} .

$$C_i = G_{i:j} + P_{i:j} C_{j-1} \quad (5.5)$$

Schematics typically show signals flowing from left to right. Arithmetic circuits break this rule because the carries flow from right to left (from the least significant column to the most significant column).



Throughout the ages, people have used many devices to perform arithmetic. Toddlers count on their fingers (and some adults stealthily do too). The Chinese and Babylonians invented the abacus as early as 2400 BC. Slide rules, invented in 1630, were in use until the 1970's, when scientific hand calculators became prevalent. Computers and digital calculators are ubiquitous today. What will be next?

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i , from A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. For a large adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N -bit adder divided into k -bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg_block} + \left(\frac{N}{k} - 1\right)t_{AND_OR} + kt_{FA} \quad (5.6)$$

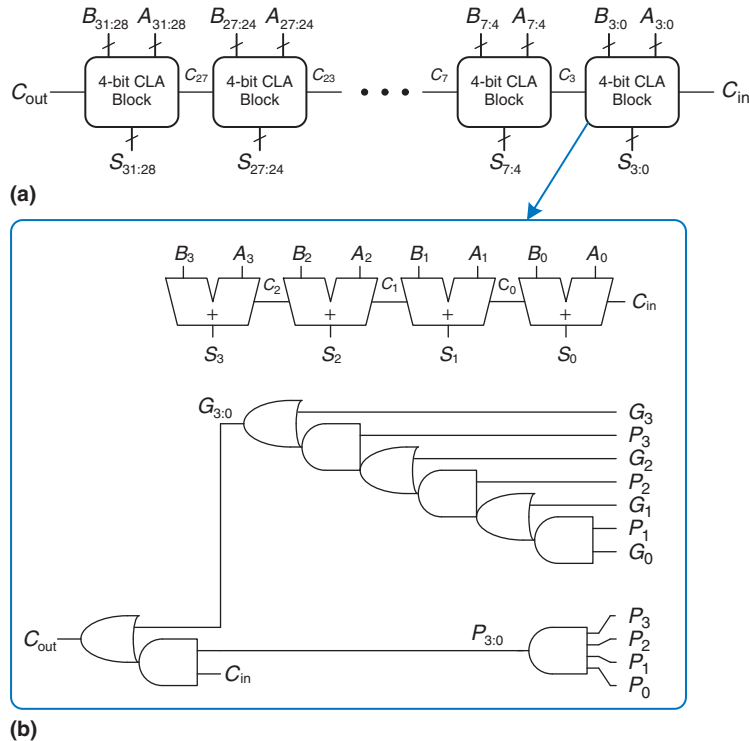


Figure 5.6 (a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block

where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate P_i and G_i , t_{pg_block} is the delay to find the generate/propagate signals P_{ij} and G_{ij} for a k -bit block, and t_{AND_OR} is the delay from C_{in} to C_{out} through the final AND/OR logic of the k -bit CLA block. For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N .

Example 5.1 RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution: According to Equation 5.1, the propagation delay of the 32-bit ripple-carry adder is $32 \times 300 \text{ ps} = 9.6 \text{ ns}$.

The CLA has $t_{pg} = 100 \text{ ps}$, $t_{pg_block} = 6 \times 100 \text{ ps} = 600 \text{ ps}$, and $t_{AND_OR} = 2 \times 100 \text{ ps} = 200 \text{ ps}$. According to Equation 5.6, the propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus $100 \text{ ps} + 600 \text{ ps} + (32/4 - 1) \times 200 \text{ ps} + (4 \times 300 \text{ ps}) = 3.3 \text{ ns}$, almost three times faster than the ripple-carry adder.

Prefix Adder*

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \quad (5.7)$$

Define column $i = -1$ to hold C_{in} , so $G_{-1} = C_{in}$ and $P_{-1} = 0$. Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i-1$ if the block spanning columns $i-1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation 5.7 as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \quad (5.8)$$

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}, G_{0:-1}, G_{1:-1}, G_{2:-1}, \dots, G_{N-2:-1}$. These signals, along with $P_{-1:-1}, P_{0:-1}, P_{1:-1}, P_{2:-1}, \dots, P_{N-2:-1}$, are called *prefixes*.

Early computers used ripple-carry adders, because components were expensive and ripple-carry adders used the least hardware. Virtually all modern PCs use prefix adders on critical paths, because transistors are now cheap and speed is of great importance.

Figure 5.7 shows an $N = 16$ -bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits $i:k$ and from the lower part spanning bits $k-1:j$. It combines these parts to form generate and propagate signals for the entire block spanning bits $i:j$ using the equations

$$G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j} \quad (5.9)$$

$$P_{i:j} = P_{i:k}P_{k-1:j} \quad (5.10)$$

In other words, a block spanning bits $i:j$ will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the upper and

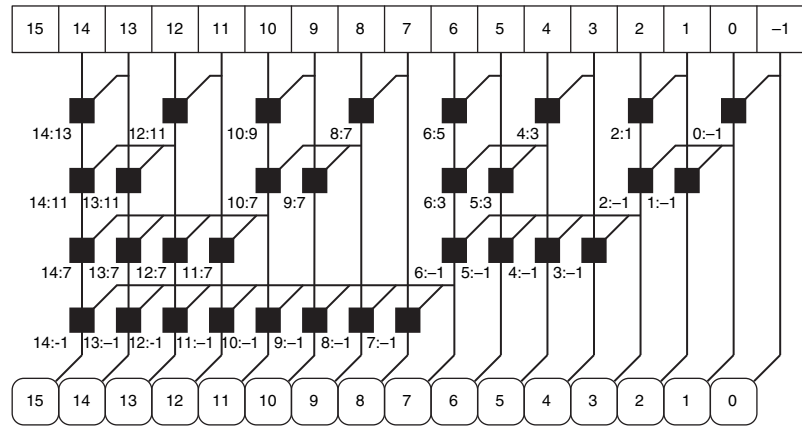
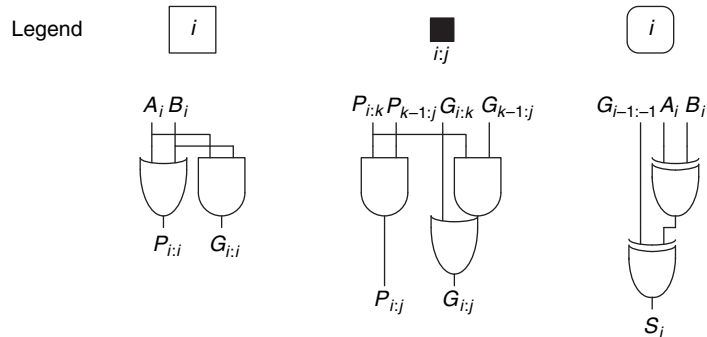


Figure 5.7 16-bit prefix adder



lower parts propagate the carry. Finally, the prefix adder computes the sums using [Equation 5.8](#).

In summary, the prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. This speedup is significant, especially for adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-lookahead adder. The network of black cells is called a *prefix tree*.

The general principle of using prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique. With some cleverness, it can be applied to many other types of circuits (see, for example, Exercise 5.7).

The critical path for an N -bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all the prefixes. $G_{i-1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i . Mathematically, the delay of an N -bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N (t_{pg_prefix}) + t_{XOR} \quad (5.11)$$

where t_{pg_prefix} is the delay of a black prefix cell.

Example 5.2 PREFIX ADDER DELAY

Compute the delay of a 32-bit prefix adder. Assume that each two-input gate delay is 100 ps.

Solution: The propagation delay of each black prefix cell t_{pg_prefix} is 200 ps (i.e., two gate delays). Thus, using [Equation 5.11](#), the propagation delay of the 32-bit prefix adder is $100 \text{ ps} + \log_2(32) \times 200 \text{ ps} + 100 \text{ ps} = 1.2 \text{ ns}$, which is about three times faster than the carry-lookahead adder and eight times faster than the ripple-carry adder from [Example 5.1](#). In practice, the benefits are not quite this great, but prefix adders are still substantially faster than the alternatives.

Putting It All Together

This section introduced the half adder, full adder, and three types of carry propagate adders: ripple-carry, carry-lookahead, and prefix adders. Faster adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate adder for a design.

Hardware description languages provide the $+$ operation to specify a CPA. Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the designer's job. [HDL Example 5.1](#) describes a CPA with carries in and out.

HDL Example 5.1 ADDER

SystemVerilog

```
module adder #(parameter N = 8)
    (input  logic [N-1:0] a, b,
     input  logic      cin,
     output logic [N-1:0] s,
     output logic      cout);

    assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
    generic(N: integer := 8);
    port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
         cin: in  STD_LOGIC;
         s:   out STD_LOGIC_VECTOR(N-1 downto 0);
         cout: out STD_LOGIC);
end;

architecture synth of adder is
    signal result: STD_LOGIC_VECTOR(N downto 0);
begin
    result <= ("0" & a) + ("0" & b) + cin;
    s      <= result(N-1 downto 0);
    cout   <= result(N);
end;
```

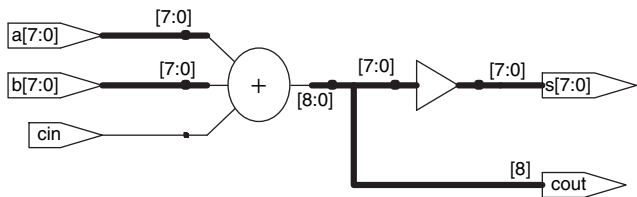


Figure 5.8 Synthesized adder

5.2.2 Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two’s complement number representation. Subtraction is almost as easy: flip the sign of the second number, then add. Flipping the sign of a two’s complement number is done by inverting the bits and adding 1.

To compute $Y = A - B$, first create the two’s complement of B : Invert the bits of B to obtain \overline{B} and add 1 to get $-B = \overline{B} + 1$. Add this quantity to A to get $Y = A + \overline{B} + 1 = A - B$. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{in} = 1$. Figure 5.9 shows the symbol for a subtractor and the underlying hardware for performing $Y = A - B$. HDL Example 5.2 describes a subtractor.

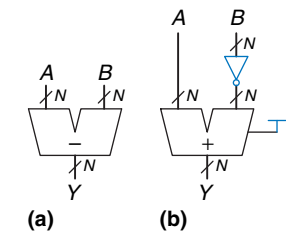


Figure 5.9 Subtractor: (a) symbol, (b) implementation

5.2.3 Comparators

A *comparator* determines whether two binary numbers are equal or if one is greater or less than the other. A comparator receives two N -bit binary numbers A and B . There are two common types of comparators.

HDL Example 5.2 SUBTRACTOR

SystemVerilog

```
module subtractor #(parameter N = 8)
    (input  logic [N-1:0] a, b,
     output logic [N-1:0] y);

    assign y = a - b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.UNSIGNED.ALL;

entity subtractor is
    generic(N: integer := 8);
    port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
         y:      out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of subtractor is
begin
    y <= a - b;
end;
```

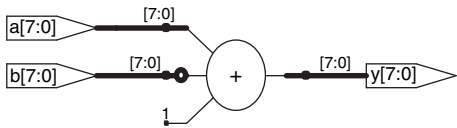


Figure 5.10 Synthesized subtractor

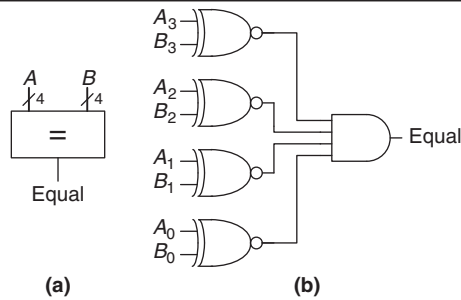


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

An *equality comparator* produces a single output indicating whether A is equal to B ($A == B$). A *magnitude comparator* produces one or more outputs indicating the relative values of A and B .

The equality comparator is the simpler piece of hardware. Figure 5.11 shows the symbol and implementation of a 4-bit equality comparator. It first checks to determine whether the corresponding bits in each column of A and B are equal using XNOR gates. The numbers are equal if all of the columns are equal.

Magnitude comparison of signed numbers is usually done by computing $A - B$ and looking at the sign (most significant bit) of the result as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B . Otherwise A is greater than or equal to B . This comparator, however, functions incorrectly upon overflow. Exercises 5.9 and 5.10 explore this limitation and how to fix it.

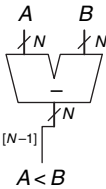


Figure 5.12 N -bit signed comparator

HDL Example 5.3 COMPARATORS

SystemVerilog

```
module comparator #(parameter N = 8)
    (input  logic [N-1:0] a, b,
     output logic eq, neq, lt, lte, gt, gte);

    assign eq  = (a == b);
    assign neq = (a != b);
    assign lt  = (a < b);
    assign lte = (a <= b);
    assign gt  = (a > b);
    assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity comparators is
    generic(N: integer := 8);
    port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
         eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparator is
begin
    eq <= '1' when (a = b) else '0';
    neq <= '1' when (a /= b) else '0';
    lt <= '1' when (a < b) else '0';
    lte <= '1' when (a <= b) else '0';
    gt <= '1' when (a > b) else '0';
    gte <= '1' when (a >= b) else '0';
end;
```

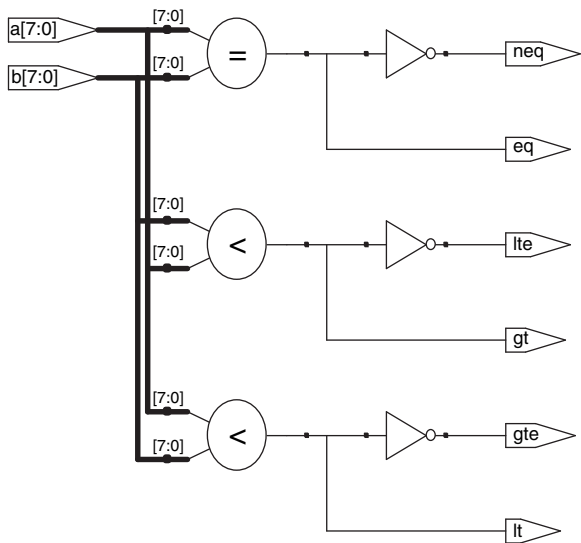


Figure 5.13 Synthesized comparators

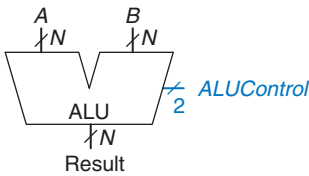


Figure 5.14 ALU symbol

HDL Example 5.3 shows how to use various comparison operations for unsigned numbers.

5.2.4 ALU

An *Arithmetic/Logical Unit (ALU)* combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU

Table 5.1 ALU operations

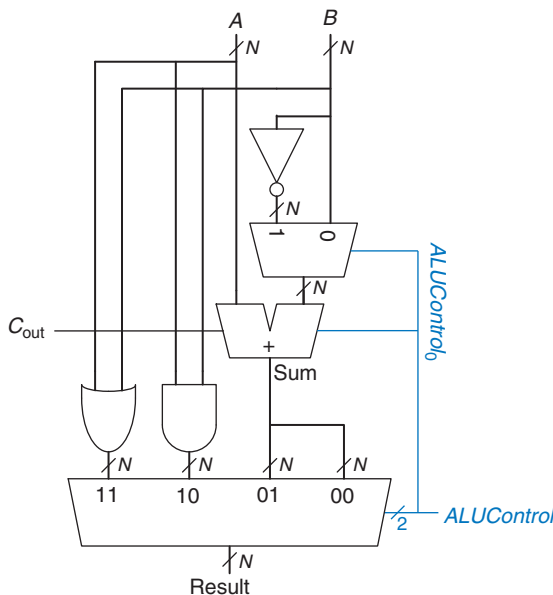
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

might perform addition, subtraction, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an N -bit ALU with N -bit inputs and outputs. The ALU receives a 2-bit control signal $ALUControl$ that specifies which function to perform. Control signals will generally be shown in blue to distinguish them from the data. Table 5.1 lists typical functions that the ALU can perform.

Figure 5.15 shows an implementation of the ALU. The ALU contains an N -bit adder and N two-input AND and OR gates. It also contains inverters and a multiplexer to invert input B when $ALUControl_0$ is asserted. A 4:1 multiplexer chooses the desired function based on $ALUControl$.

More specifically, if $ALUControl = 00$, the output multiplexer chooses $A + B$. If $ALUControl = 01$, the ALU computes $A - B$. (Recall from Section 5.2.2 that $\bar{B} + 1 = -B$ in two's complement arithmetic. Because $ALUControl_0$ is 1, the adder receives inputs A and \bar{B} and an asserted carry in, causing

**Figure 5.15** N -bit ALU

Architecture

6

6.1 INTRODUCTION

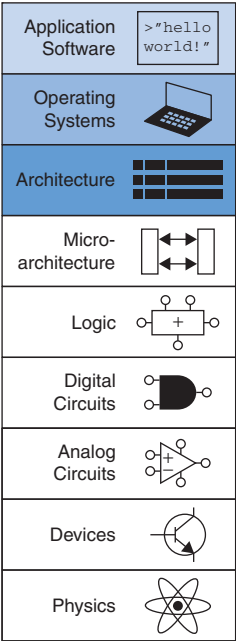
The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and branch. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language. Just as we use letters to encode human language, computers use binary numbers to encode machine language. The ARM architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called assembly language.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and branch, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

- 6.1 Introduction
- 6.2 Assembly Language
- 6.3 Programming
- 6.4 Machine Language
- 6.5 Lights, Camera, Action: Compiling, Assembling, and Loading*
- 6.6 Odds and Ends*
- 6.7 Evolution of ARM Architecture
- 6.8 Another Perspective: x86 Architecture
- 6.9 Summary
- Exercises
- Interview Questions



A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the microarchitecture and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

The “ARM architecture” we describe is ARM version 4 (ARMv4), which forms the core of the instruction set. [Section 6.7](#) summarizes new features in versions 5–8 of the architecture. The *ARM Architecture Reference Manual* (ARM), available online, is the authoritative definition of the architecture.

In this text, we introduce the ARM architecture. This architecture was first developed in the 1980s by Acorn Computer Group, which spun off Advanced RISC Machines Ltd., now known as ARM. Over 10 billion ARM processors are sold every year. Almost all cell phones and tablets contain multiple ARM processors. The architecture is used in everything from pinball machines to cameras to robots to cars to rack-mounted servers. ARM is unusual in that it does not sell processors directly, but rather licenses other companies to build its processors, often as part of a larger system-on-chip. For example, Samsung, Altera, Apple, and Qualcomm all build ARM processors, either using microarchitectures purchased from ARM or microarchitectures developed internally under license from ARM. We choose to focus on ARM because it is a commercial leader and because the architecture is clean, with few idiosyncrasies. We start by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

Throughout the chapter, we motivate the design of the ARM architecture using four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design*: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer’s native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the ARM instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java.

(These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables *b* and *c* and writing the result to *a*. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in ARM assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	ARM Assembly Code
<code>a = b + c;</code>	<code>ADD a, b, c</code>

The first part of the assembly instruction, `ADD`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on *b* and *c*, the *source operands*, and the result is written to *a*, the *destination operand*.

Code Example 6.2 SUBTRACTION

High-Level Code	ARM Assembly Code
<code>a = b - c;</code>	<code>SUB a, b, c</code>

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the `ADD` instruction except for the operation specification, `SUB`. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple ARM instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with `//` and continue until the end of the line. Multiline comments begin with `/*` and end with `*/`. In ARM assembly language, only single-line comments

We used Keil’s ARM Microcontroller Development Kit (MDK-ARM) to compile, assemble, and simulate the example assembly code in this chapter. The MDK-ARM is a free development tool that comes with a complete ARM compiler. Labs available on this textbook’s companion site (see Preface) show how to install and use this tool to write, compile, simulate, and debug both C and assembly programs.

Mnemonic (pronounced ni-mon-ik) comes from the Greek word *μνηΕσκεσται*, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0’s and 1’s representing the same operation.

Code Example 6.3 MORE COMPLEX CODE

High-Level Code	ARM Assembly Code
<pre>a = b + c - d; // single-line comment /* multiple-line comment */</pre>	<pre>ADD t, b, c ; t = b + c SUB a, t, d ; a = t - d</pre>

are used. They begin with a semicolon (;) and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable `t` to store the intermediate result. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The ARM instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, ARM is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel’s x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a “string move” instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation. An instruction set with 256 complex instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1, the variables `a`, `b`, and `c` are all operands. But computers operate on 1’s and 0’s, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use

various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Version 8 of the ARM architecture has been extended to 64 bits, but we will focus on the 32-bit version in this book.

Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The ARM architecture uses 16 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small register file is faster than reading it from a large memory. A register file is typically built from a small SRAM array (see Section 5.5.3).

Code Example 6.4 shows the `ADD` instruction with register operands. ARM register names are preceded by the letter 'R'. The variables `a`, `b`, and `c` are arbitrarily placed in `R0`, `R1`, and `R2`. The name `R1` is pronounced “register 1” or “R1” or “register R1”. The instruction adds the 32-bit values contained in `R1` (`b`) and `R2` (`c`) and writes the 32-bit result to `R0` (`a`). Code Example 6.5 shows ARM assembly code using a register, `R4`, to store the intermediate calculation of `b + c`:

Code Example 6.4 REGISTER OPERANDS

High-Level Code	ARM Assembly Code
<code>a = b + c;</code>	<code>; R0 = a, R1 = b, R2 = c</code> <code>ADD R0, R1, R2 ; a = b + c</code>

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	ARM Assembly Code
<code>a = b + c - d;</code>	<code>; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t</code> <code>ADD R4, R1, R2 ; t = b + c</code> <code>SUB R0, R4, R3 ; a = t - d</code>

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into ARM assembly language. Assume variables a–c are held in registers R0–R2 and f–j are in R3–R7.

```
a = b - c ;  
f = (g + h) - (i + j) ;
```

Solution: The program uses four assembly language instructions.

```
; ARM assembly code  
; R0 = a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j  
SUB R0, R1, R2      ; a = b - c  
ADD R8, R4, R5      ; R8 = g + h  
ADD R9, R6, R7      ; R9 = i + j  
SUB R3, R8, R9      ; f = (g + h) - (i + j)
```

The Register Set

Table 6.1 lists the name and use for each of the 16 ARM registers. R0–R12 are used for storing variables; R0–R3 also have special uses during procedure calls. R13–R15 are also called SP, LR, and PC, and they will be described later in this chapter.

Constants/Immediates

In addition to register operations, ARM instructions can use constant or *immediate* operands. These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the ADD instruction adding an immediate to a register. In assembly code, the immediate is preceded by the # symbol and can be written in decimal or hexadecimal. Hexadecimal constants in ARM assembly language start with 0x, as they

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code	ARM Assembly Code
<pre>a = a + 4; b = a - 12;</pre>	<pre>; R7 = a, R8 = b ADD R7, R7, #4 ; a = a + 4 SUB R8, R7, #0xc ; b = a - 12</pre>

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code	ARM Assembly Code
<pre>i = 0; x = 4080;</pre>	<pre>; R4 = i, R5 = x MOV R4, #0 ; i = 0 MOV R5, #0xff0 ; x = 4080</pre>

do in C. immediates are unsigned 8- to 12-bit numbers with a peculiar encoding described in [Section 6.4](#).

The move instruction (MOV) is a useful way to initialize register values. Code Example 6.7 initializes the variables `i` and `x` to 0 and 4080, respectively. MOV can also take a register source operand. For example, `MOV R1, R7` copies the contents of register R7 into R1.

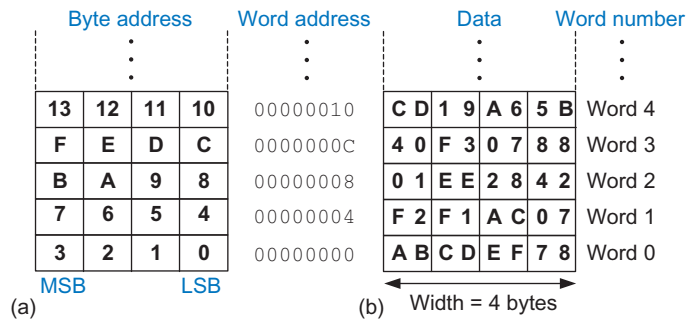
Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 15 variables. However, data can also be stored in memory. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the ARM architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from Section 5.5 that memories are organized as an array of data words. The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in [Figure 6.1\(a\)](#). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4. The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Both the 32-bit word address and the data value in [Figure 6.1\(b\)](#) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

ARM provides the *load register* instruction, `LDR`, to read a data word from memory into a register. Code Example 6.8 loads memory word 2 into a (R7). In C, the number inside the brackets is the *index* or word number,

Figure 6.1 ARM byte-addressable memory showing: (a) byte address and (b) data



Code Example 6.8 READING MEMORY

High-Level Code	ARM Assembly Code
<pre>a = mem[2];</pre>	<pre>; R7 = a MOV R5, #0 ; base address = 0 LDR R7, [R5, #8] ; R7 <= data at memory address (R5+8)</pre>

A read from the base address (i.e., index 0) is a special case that requires no offset in the assembly code. For example, a memory read from the base address held in R5 is written as `LDR R3, [R5]`.

ARMv4 requires *word-aligned addresses* for LDR and STR, that is, a word address that is divisible by four. Since ARMv6, this alignment restriction can be removed by setting a bit in the ARM system control register, but performance of *unaligned* loads is usually worse. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others, such as MIPS, require strict alignment for simplicity. Of course, byte addresses for load byte and store byte, LDRB and STRB (discussed in Section 6.3.6), need not be word aligned.

which we discuss further in Section 6.3.6. The LDR instruction specifies the memory address using a *base register* (R5) and an *offset* (8). Recall that each data word is 4 bytes, so word number 1 is at address 4, word number 2 is at address 8, and so on. The word address is four times the word number. The memory address is formed by adding the contents of the base register (R5) and the offset. ARM offers several modes for accessing memory, as will be discussed in Section 6.3.6.

After the load register instruction (LDR) is executed in Code Example 6.8, R7 holds the value 0x01EE2842, which is the data value stored at memory address 8 in Figure 6.1.

ARM uses the *store register* instruction, STR, to write a data word from a register into memory. Code Example 6.9 writes the value 42 from register R9 into memory word 5.

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.2. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word

Code Example 6.9 WRITING MEMORY

High-Level Code	ARM Assembly Code
<pre>mem[5] = 42;</pre>	<pre>MOV R1, #0 ; base address = 0 MOV R9, #42 STR R9, [R1, #0x14] ; value stored at memory address (R1+20) = 42</pre>

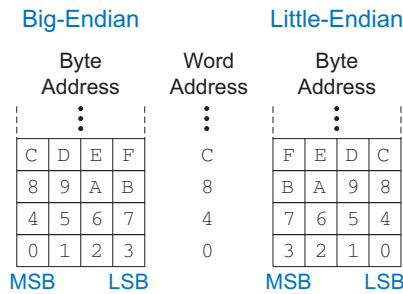


Figure 6.2 Big-endian and little-endian memory addressing

differ. In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. ARM prefers little-endian but provides support in some versions for *bi-endian* data addressing, which allows data loads and stores in either format. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.

6.3 PROGRAMMING

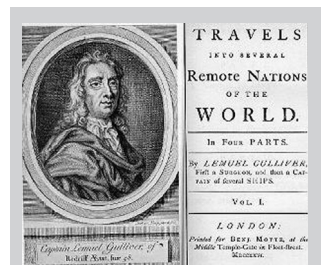
Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, conditional execution, if/else statements, for and while loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into ARM assembly code.

6.3.1 Data-processing Instructions

The ARM architecture defines a variety of *data-processing* instruction (often called logical and arithmetic instructions in other architectures). We introduce these instructions briefly here because they are necessary to implement higher-level constructs. Appendix B provides a summary of ARM instructions.

Logical Instructions

ARM *logical operations* include AND, ORR (OR), EOR (XOR), and BIC (bit clear). These each operate bitwise on two sources and write the result



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories, the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

These terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, Leeds University Library.)

Figure 6.3 Logical operations

		Source registers			
R1		0100 0110	1010 0001	1111 0001	1011 0111
R2		1111 1111	1111 1111	0000 0000	0000 0000
Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

to a destination register. The first source is always a register and the second source is either an immediate or another register. Another logical operation, MVN (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register. Figure 6.3 shows examples of these operations on the two source values 0x46A1F1B7 and 0xFFFF0000. The figure shows the values stored in the destination register after the instruction executes.

The bit clear (BIC) instruction is useful for masking bits (i.e., forcing unwanted bits to 0). BIC R6, R1, R2 computes R1 AND NOT R2. In other words, BIC clears the bits that are asserted in R2. In this case, the top two bytes of R1 are cleared or *masked*, and the unmasked bottom two bytes of R1, 0xF1B7, are placed in R6. Any subset of register bits can be masked.

The ORR instruction is useful for combining bitfields from two registers. For example, 0x347A0000 ORR 0x000072FC = 0x347A72FC.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. The rotate instruction rotates the value in a register right by up to 31 bits. We refer to both shift and rotate generically as shift operations. ARM shift operations are LSL (logical shift left), LSR (logical shift right), ASR (arithmetic shift right), and ROR (rotate right). There is no ROL instruction because left rotation can be performed with a right rotation by a complementary amount.

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0’s. However, right shifts can be either logical (0’s shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). The amount by which to shift can be an immediate or a register.

Figure 6.4 shows the assembly code and resulting register values for LSL, LSR, ASR, and ROR when shifting by an immediate value. R5 is shifted by the immediate amount, and the result is placed in the destination register.

Source register				
R5	1111 1111	0001 1100	0001 0000	1110 0111
Result				
LSL R0, R5, #7 R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17 R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3 R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21 R3	1110 0000	1000 0111	0011 1111	1111 1000

Figure 6.4 Shift instructions with immediate shift amounts

Source registers				
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100
Result				
LSL R4, R8, R6 R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6 R5	1100 0001	0110 1110	0111 0000	1000 0001

Figure 6.5 Shift instructions with register shift amounts

Shifting a value left by N is equivalent to multiplying it by 2^N . Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N , as discussed in Section 5.2.5. Logical shifts are also used to extract or assemble bitfields.

Figure 6.5 shows the assembly code and resulting register values for shift operations where the shift amount is held in a register, R6. This instruction uses the *register-shifted register* addressing mode, where one register (R8) is shifted by the amount (20) held in a second register (R6).

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. The ARM architecture provides *multiply instructions* that result in a 32-bit or 64-bit product. Multiply (MUL) multiplies two 32-bit numbers and produces a 32-bit result. MUL R1, R2, R3 multiplies the values in R2 and R3 and places the least significant bits of the product in R1; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose result fits in 32 bits. UMULL (unsigned multiply long) and SMULL (signed multiply long) multiply two 32-bit numbers and produce a 64-bit product. For example, UMULL R1, R2, R3, R4 performs an unsigned multiply of R3 and R4. The least significant 32 bits of the product is placed in R1 and the most significant 32 bits are placed in R2.

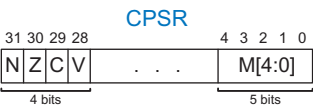


Figure 6.6 Current Program Status Register (CPSR)

The least significant five bits of the CPSR are *mode* bits and will be described in [Section 6.6.3](#).

Other useful instructions for comparing two values are CMN, TST, and TEQ. Each instruction performs an operation, updates the condition flags, and discards the result. CMN (compare negative) compares the first source to the negative of the second source by adding the two sources. As will be shown in [Section 6.4](#), ARM instructions only encode positive immediates. So, CMN R2, #20 is used instead of CMP R2, #-20. TST (test) ANDs the source operands. It is useful for checking if some portion of the register is zero or nonzero. For example, TST R2, #0xFF would set the Z flag if the low byte of R2 is 0. TEQ (test if equal) checks for equivalence by XOR-ing the sources. Thus, the Z flag is set when they are equal and the N flag is set when the signs are different.

Each of these instructions also has a multiply-accumulate variant, MLA, SMLAL, and UMLAL, that adds the product to a running 32- or 64-bit sum. These instructions can boost the math performance in applications such as matrix multiplication and signal processing consisting of repeated multiplies and adds.

6.3.2 Condition Flags

Programs would be boring if they could only run in the same order every time. ARM instructions optionally set *condition flags* based on whether the result is negative, zero, etc. Subsequent instructions then execute *conditionally*, depending on the state of those condition flags. The ARM condition flags, also called *status flags*, are negative (N), zero (Z), carry (C), and overflow (V), as listed in [Table 6.2](#). These flags are set by the ALU (see [Section 5.2.4](#)) and are held in the top 4 bits of the 32-bit *Current Program Status Register (CPSR)*, as shown in [Figure 6.6](#).

The most common way to set the status bits is with the compare (CMP) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the Z flag is set. If the first number is an unsigned value that is higher than or the same as the second, the subtraction will produce a carry out and the C flag is set.

Subsequent instructions can conditionally execute depending on the state of the flags. The instruction mnemonic is followed by a *condition mnemonic* that indicates when to execute. [Table 6.3](#) lists the 4-bit condition field (*cond*), the condition mnemonic, name, and the state of the condition flags that result in instruction execution (CondEx). For example, suppose a program performs CMP R4, R5, and then ADDEQ R1, R2, R3. The compare sets the Z flag if R4 and R5 are equal, and the ADDEQ executes only if the Z flag is set. The *cond* field will be used in machine language encodings in [Section 6.4](#).

Table 6.2 Condition flags

Flag	Name	Description
N	Negative	Instruction result is negative, i.e., bit 31 of the result is 1
Z	Zero	Instruction result is zero
C	Carry	Instruction causes a carry out
V	oVerflow	Instruction causes an overflow

Table 6.3 Condition mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\bar{Z}
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	\bar{C}
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	\bar{N}
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	\bar{V}
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
1010	GE	Signed greater than or equal	$\bar{N} \oplus \bar{V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\bar{N} \oplus \bar{V})$
1101	LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by “S.” For example, SUBS R2, R3, R7 will subtract R7 from R3, put the result in R2, and set the condition flags. Table B.5 in Appendix B summarizes which condition flags are influenced by each instruction. All data-processing instructions will affect the N and Z flags based on whether the result is zero or has the most significant bit set. ADDS and SUBS also influence V and C , and shifts influence C .

Code Example 6.10 shows instructions that execute conditionally. The first instruction, CMP R2, R3, executes unconditionally and sets the condition flags. The remaining instructions execute conditionally, depending on the values of the condition flags. Suppose R2 and R3 contain the values 0x80000000 and 0x00000001. The compare computes $R2 - R3 = 0x80000000 - 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF$ with a carry out ($C=1$). The sources had opposite signs and the sign of the result differs from the sign of the first source, so the result overflows ($V=1$). The remaining flags (N and Z) are 0. ANDHS executes

Condition mnemonics differ for signed and unsigned comparison. For example, ARM provides two forms of greater than or equal comparison: HS (CS) is used for unsigned numbers and GE for signed. For unsigned numbers, $A - B$ will produce a carry out (C) when $A \geq B$. For signed numbers, $A - B$ will make N and V either both 0 or both 1 when $A \geq B$. Figure 6.7 highlights the difference between HS and GE comparisons with two examples using 4-bit numbers for ease of interpretation.

	Unsigned	Signed
A = 1001₂	A = 9	A = -7
B = 0010₂	B = 2	B = 2
A - B: 1001	NZCV = 0011 ₂	
+ 1110	HS: TRUE	
(a) 10111	GE: FALSE	

	Unsigned	Signed
A = 0101₂	A = 5	A = 5
B = 1101₂	B = 13	B = -3
A - B: 0101	NZCV = 1001 ₂	
+ 0011	HS: FALSE	
(b) 1000	GE: TRUE	

Figure 6.7 Signed vs. unsigned comparison: HS vs. GE

Code Example 6.10 CONDITIONAL EXECUTION**ARM Assembly Code**

```
CMP    R2, R3
ADDEQ  R4, R5, #78
ANDHS  R7, R8, R9
ORRMI  R10, R11, R12
EORLT  R12, R7, R10
```

because $C = 1$. EORLT executes because N is 0 and V is 1 (see [Table 6.3](#)). Intuitively, ANDHS and EORLT execute because $R2 \geq R3$ (unsigned) and $R2 < R3$ (signed), respectively. ADDEQ and ORRMI do not execute because the result of $R2 - R3$ is not zero (i.e., $R2 \neq R3$) or negative.

6.3.3 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, if/else statements, switch/case statements, while loops, and for loops all conditionally execute code depending on some test.

One way to make decisions is to use conditional execution to ignore certain instructions. This works well for simple if statements where a small number of instructions are ignored, but it is wasteful for if statements with many instructions in the body, and it is insufficient to handle loops. Thus, ARM and most other architectures use *branch instructions* to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 after each instruction to point to the next instruction. (Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.) Branch instructions change the program counter. ARM includes two types of branches: a simple *branch* (B) and *branch and link* (BL). BL is used for function calls and is discussed in [Section 6.3.7](#). Like other ARM instructions, branches can be unconditional or conditional. Branches are also called *jumps* in some architectures.

Code Example 6.11 shows unconditional branching using the branch instruction B. When the code reaches the B TARGET instruction, the branch is *taken*. That is, the next instruction executed is the SUB instruction just after the *label* called TARGET.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses (see [Section 6.4.3](#)). ARM assembly labels cannot be reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help

Code Example 6.11 UNCONDITIONAL BRANCHING**ARM Assembly Code**

```

ADD R1, R2, #17    ; R1 = R2 + 17
B   TARGET         ; branch to TARGET
ORR R1, R1, R3     ; not executed
AND R3, R1, #0xFF  ; not executed

TARGET
SUB R1, R1, #78    ; R1 = R1 - 78

```

Code Example 6.12 CONDITIONAL BRANCHING**ARM Assembly Code**

```

MOV R0, #4         ; R0 = 4
ADD R1, R0, R0     ; R1 = R0 + R0 = 8
CMP R0, R1         ; set flags based on R0-R1 = -4. NZCV = 1000
BEQ THERE         ; branch not taken (Z != 1)
ORR R1, R1, #1     ; R1 = R1 OR 1 = 9

THERE
ADD R1, R1, #78    ; R1 = R1 + 78 = 87

```

make labels stand out. The ARM compiler makes this a requirement: labels must not be indented, and instructions must be preceded by white space. Some compilers, including GCC, require a colon after the label.

Branch instructions can execute conditionally based on the condition mnemonics listed in [Table 6.3](#). Code Example 6.12 illustrates the use of BEQ, branching dependent on equality ($Z = 1$). When the code reaches the BEQ instruction, the Z condition flag is 0 (i.e., $R0 \neq R1$), so the branch is *not taken*. That is, the next instruction executed is the ORR instruction.

6.3.4 Conditional Statements

if, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into ARM assembly language.

if Statements

An if statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.13 shows how to translate an if statement into ARM assembly code.

Code Example 6.13 IF STATEMENT

High-Level Code	ARM Assembly Code
<pre>if (apples == oranges) f = i + 1; f = f - i;</pre>	<pre>; R0 = apples, R1 = oranges, R2 = f, R3 = i CMP R0, R1 ; apples == oranges ? BNE L1 ; if not equal, skip if block ADD R2, R3, #1 ; if block: f = i + 1 L1 SUB R2, R2, R3 ; f = f - i</pre>

Recall that != is an inequality comparison and == is an equality comparison in the high-level code.

The assembly code for the if statement tests the opposite condition of the one in the high-level code. In Code Example 6.13, the high-level code tests for apples == oranges. The assembly code tests for apples != oranges using BNE to skip the if block if the condition is **not** satisfied. Otherwise, apples == oranges, the branch is not taken, and the if block is executed.

Because any instruction can be conditionally executed, the ARM assembly code for Code Example 6.13 could also be written more compactly as shown below.

```
CMP    R0, R1      ; apples == oranges ?
ADDEQ  R2, R3, #1  ; f = i + 1 on equality (i.e., Z = 1)
SUB    R2, R2, R3  ; f = f - i
```

This solution with conditional execution is shorter and also faster because it involves one fewer instruction. Moreover, we will see in Section 7.5.3 that branches sometimes introduce extra delay, whereas conditional execution is always fast. This example shows the power of conditional execution in the ARM architecture.

In general, when a block of code has a single instruction, it is better to use conditional execution rather than branch around it. As the block becomes longer, the branch becomes valuable because it avoids wasting time fetching instructions that will not be executed.

if/else Statements

if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.14 shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In Code Example 6.14, the high-level code tests for apples == oranges, and the assembly code tests for apples != oranges. If that opposite condition is TRUE, BNE skips the if block and executes the else block. Otherwise, the if block executes and finishes with an unconditional branch (B) past the else block.

Code Example 6.14 IF/ELSE STATEMENT

High-Level Code	ARM Assembly Code
<pre> if (apples == oranges) f = i + 1; else f = f - i; </pre>	<pre> ; R0 = apples, R1 = oranges, R2 = f, R3 = i CMP R0, R1 ; apples == oranges? BNE L1 ; if not equal, skip if block ADD R2, R3, #1 ; if block: f = i + 1 B L2 ; skip else block L1 SUB R2, R2, R3 ; else block: f = f - i L2 </pre>

Again, because any instruction can conditionally execute and because the instructions within the if block do not change the condition flags, the ARM assembly code for Code Example 6.14 could also be written much more succinctly as:

```

CMP    R0, R1      ; apples == oranges?
ADDEQ R2, R3, #1   ; f = i + 1 on equality (i.e., Z = 1)
SUBNE R2, R2, R3   ; f = f - i on not equal (i.e., Z = 0)

```

switch/case Statements*

switch/case statements execute one of several blocks of code depending on the conditions. If no conditions are met, the *default block* is executed. A case statement is equivalent to a series of *nested if/else* statements. Code Example 6.15 shows two high-level code snippets with the same

Code Example 6.15 SWITCH/CASE STATEMENT

High-Level Code	ARM Assembly Code
<pre> switch (button) { case 1: amt = 20; break; case 2: amt = 50; break; case 3: amt = 100; break; default: amt = 0; } // equivalent function using // if/else statements if (button == 1) amt = 20; else if (button == 2) amt = 50; else if (button == 3) amt = 100; else amt = 0; </pre>	<pre> ; R0 = button, R1 = amt CMP R0, #1 ; is button 1 ? MOVEQ R1, #20 ; amt = 20 if button is 1 BEQ DONE ; break CMP R0, #2 ; is button 2 ? MOVEQ R1, #50 ; amt = 50 if button is 2 BEQ DONE ; break CMP R0, #3 ; is button 3 ? MOVEQ R1, #100 ; amt = 100 if button is 3 BEQ DONE ; break MOV R1, #0 ; default amt = 0 DONE </pre>

functionality: they calculate whether to dispense \$20, \$50, or \$100 from an ATM (automatic teller machine) depending on the button pressed. The ARM assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. while loops and for loops are common loop constructs used by high-level languages. This section shows how to translate them into ARM assembly language, taking advantage of conditional branching.

while Loops

while loops repeatedly execute a block of code until a condition is *not* met. The while loop in Code Example 6.16 determines the value of *x* such that $2^x = 128$. It executes seven times, until *pow* = 128.

Like if/else statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, *R0* == 128), the while loop is finished. If not (*R0* ≠ 128), the branch isn't taken and the loop body executes.

The `int` data type in C refers to a word of data representing a two's complement integer. ARM uses 32-bit words, so an `int` represents a number in the range $[-2^{31}, 2^{31} - 1]$.

Code Example 6.16 WHILE LOOP

High-Level Code	ARM Assembly Code
<pre>int pow = 1; int x = 0; while (pow != 128) { pow = pow * 2; x = x + 1; }</pre>	<pre>; R0 = pow, R1 = x MOV R0, #1 ; pow = 1 MOV R1, #0 ; x = 0 WHILE CMP R0, #128 ; pow != 128 ? BEQ DONE ; if pow == 128, exit loop LSL R0, R0, #1 ; pow = pow * 2 ADD R1, R1, #1 ; x = x + 1 B WHILE ; repeat loop DONE</pre>

In Code Example 6.16, the while loop compares *pow* to 128 and exits the loop if it is equal. Otherwise it doubles *pow* (using a left shift), increments *x*, and branches back to the start of the while loop.

for Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time through the while loop. for loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The format of the for loop is:

```
for (initialization; condition; loop operation)
    statement
```


Code Example 6.17 FOR LOOP

High-Level Code	ARM Assembly Code
<pre>int i; int sum = 0; for (i = 0; i < 10; i = i + 1) { sum = sum + i; }</pre>	<pre>; R0 = i, R1 = sum MOV R1, #0 ; sum = 0 MOV R0, #0 ; i = 0 loop initialization FOR CMP R0, #10 ; i < 10 ? check condition BGE DONE ; if (i >= 10) exit loop ADD R1, R1, R0 ; sum = sum + i loop body ADD R0, R0, #1 ; i = i + 1 loop operation B FOR ; repeat loop DONE</pre>

The initialization code executes before the for loop begins. The condition is tested at the beginning of each loop. If the condition is not met, the loop exits. The loop operation executes at the end of each loop.

Code Example 6.17 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as `i` is less than 10. Note that this example also illustrates relative comparisons. The loop checks the `<` condition to continue, so the assembly code checks the opposite condition, `>=`, to exit the loop.

Loops are especially useful for accessing large amounts of similar data stored in memory, which is discussed next.

6.3.6 Memory

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array.

Figure 6.8 shows a 200-element array of scores stored in memory. Code Example 6.18 is a grade inflation algorithm that adds 10 points to each of the scores. Note that the code for initializing the scores array is not shown. The index into the array is a variable (`i`) rather than a constant, so we must multiply it by 4 before adding it to the base address.

ARM can *scale* (multiply) the index, add it to the base address, and load from memory in a single instruction. Instead of the LSL and LDR instruction sequence in Code Example 6.18, we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

`R1` is scaled (shifted left by two) then added to the base address (`R0`). Thus, the memory address is $R0 + (R1 \times 4)$.



Figure 6.8 Memory holding scores[200] starting at base address 0x14000000

Code Example 6.18 ACCESSING ARRAYS USING A FOR LOOP

High-Level Code	ARM Assembly Code
<pre>int i; int scores[200]; ... for (i = 0; i < 200; i = i + 1) scores[i] = scores[i] + 10;</pre>	<pre>; R0 = array base address, R1 = i ; initialization code ... MOV R0, #0x14000000 ; R0 = base address MOV R1, #0 ; i = 0 LOOP CMP R1, #200 ; i < 200? BGE L3 ; if i ≥ 200, exit loop LSL R2, R1, #2 ; R2 = i * 4 LDR R3, [R0, R2] ; R3 = scores[i] ADD R3, R3, #10 ; R3 = scores[i] + 10 STR R3, [R0, R2] ; scores[i] = scores[i] + 10 ADD R1, R1, #1 ; i = i + 1 B LOOP ; repeat loop L3</pre>

In addition to scaling the index register, ARM provides offset, pre-indexed, and post-indexed addressing to enable dense and efficient code for array accesses and function calls. Table 6.4 gives examples of each indexing mode. In each case, the base register is R1 and the offset is R2. The offset can be subtracted by writing $-R2$. The offset may also be an immediate in the range of 0–4095 that can be added (e.g., #20) or subtracted (e.g., #–20).

Offset addressing calculates the address as the base register \pm the offset; the base register is unchanged. *Pre-indexed addressing* calculates the address as the base register \pm the offset and updates the base register to this new address. *Post-indexed addressing* calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register \pm the offset. We have seen many examples of offset indexing mode. Code Example 6.19 shows the for loop from Code Example 6.18 rewritten to use post-indexing, eliminating the ADD to increment i .

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	$R1 + R2$	Unchanged
Pre-index	LDR R0, [R1, R2]!	$R1 + R2$	$R1 = R1 + R2$
Post-index	LDR R0, [R1], R2	R1	$R1 = R1 + R2$

Code Example 6.19 FOR LOOP USING POST-INDEXING

High-Level Code	ARM Assembly Code
<pre>int i; int scores[200]; ... for (i = 0; i < 200; i = i + 1) scores[i] = scores[i] + 10;</pre>	<pre>; R0 = array base address ; initialization code ... MOV R0, #0x14000000 ; R0 = base address ADD R1, R0, #800 ; R1 = base address + (200*4) LOOP CMP R0, R1 ; reached end of array? BGE L3 ; if yes, exit loop LDR R2, [R0] ; R2 = scores[i] ADD R2, R2, #10 ; R2 = scores[i] + 10 STR R2, [R0], #4 ; scores[i] = scores[i] + 10 ; then R0 = R0 + 4 B LOOP ; repeat loop L3</pre>

Bytes and Characters

Numbers in the range [−128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange (ASCII)*, which assigns each text character a unique byte value. Table 6.5 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

ARM provides load byte (LDRB), load signed byte (LDRSB), and store byte (STRB) to access individual bytes in memory. LDRB zero-extends the byte, whereas LDRSB sign-extends the byte to fill the entire 32-bit register. STRB stores the least significant byte of the 32-bit register into the specified byte address in memory. All three are illustrated in Figure 6.9, with

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see www.unicode.org.

LDRH, LDRSH, and STRH are similar, but access 16-bit *halfwords*.

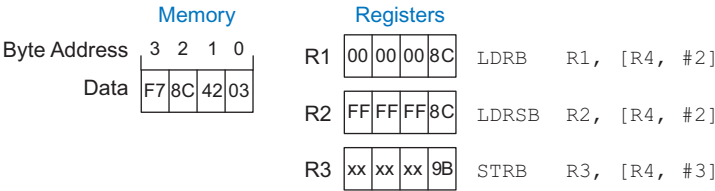


Figure 6.9 Instructions for loading and storing bytes

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as – . – . . . , and – . . . , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001.

However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

Table 6.5 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	–	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

the base address R4 being 0. LDRB loads the byte at memory address 2 into the least significant byte of R1 and fills the remaining register bits with 0. LDRSB loads this byte into R2 and sign-extends the byte into the upper 24 bits of the register. STRB stores the least significant byte of R3 (0x9B) into memory byte 3; it replaces 0xF7 with 0x9B. The more significant bytes of R3 are ignored.

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, [Figure 6.10](#) shows the string “Hello!” (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long

Example 6.2 USING LDRB AND STRB TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into ARM assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that R0 already holds the base address of chararray.

```
// high-level code
// chararray[10] declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
    chararray[i] = chararray[i] - 32;
```

Solution:

```
; ARM assembly code
; R0 = base address of chararray (initialized earlier), R1 = i
MOV    R1, #0           ; i = 0
LOOP   CMP    R1, #10    ; i < 10 ?
       BGE    DONE      ; if (i >= 10), exit loop
       LDRB   R2, [R0, R1] ; R2 = mem[R0+R1] = chararray[i]
       SUB    R2, R2, #32 ; R2 = chararray[i] - 32
       STRB   R2, [R0, R1] ; chararray[i] = R2
       ADD    R1, R1, #1  ; i = i + 1
       B      LOOP       ; repeat loop
DONE
```

and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H=0x48) is stored at the lowest byte address (0x1522FFF0).

6.3.7 Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call,

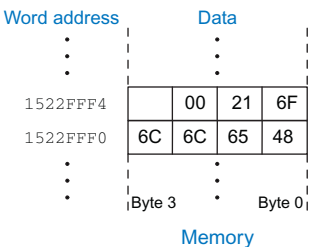


Figure 6.10 The string “Hello!” stored in memory

and the callee places the return value in register R0 before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the link register LR at the same time it jumps to the callee using the branch and link instruction (BL). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (R4–R11, and LR) and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It shows how functions access input arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

ARM uses the branch and link instruction (BL) to call a function and moves the link register to the PC (MOV PC, LR) to return from a function. Code Example 6.20 shows the main function calling the simple function. main is the caller, and simple is the callee. The simple function is called with no input arguments and generates no return value; it just returns to the caller. In Code Example 6.20, instruction addresses are given to the left of each ARM instruction in hexadecimal.

BL (branch and link) and MOV PC, LR are the two essential instructions needed for a function call and return. BL performs two tasks: it stores the *return address* of the next instruction (the instruction

Code Example 6.20 simple FUNCTION CALL

High-Level Code	ARM Assembly Code
<pre>int main() { simple(); ... } // void means the function returns no value void simple() { return; }</pre>	<pre>0x00008000 MAIN 0x00008020 BL SIMPLE ; call the simple function 0x0000902C SIMPLE MOV PC, LR ; return</pre>

after BL) in the link register (LR), and it branches to the target instruction.

In Code Example 6.20, the `main` function calls the `simple` function by executing the branch and link instruction (BL). BL branches to the `SIMPLE` label and stores `0x00008024` in LR. The `simple` function returns immediately by executing the instruction `MOV PC, LR`, copying the return address from the LR back to the PC. The `main` function then continues executing at this address (`0x00008024`).

Input Arguments and Return Values

The `simple` function in Code Example 6.20 receives no input from the calling function (`main`) and returns no output. By ARM convention, functions use R0–R3 for input arguments and R0 for the return value. In Code Example 6.21, the function `diffofsums` is called with four arguments and returns one result. `result` is a local variable, which we choose to keep in R4.

According to ARM convention, the calling function, `main`, places the function arguments from left to right into the input registers, R0–R3. The called function, `diffofsums`, stores the return value in the return register, R0. When a function with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.21 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main() {
    int y;
    . . .
    y = diffofsums(2, 3, 4, 5);
    . . .
}

int diffofsums(int f, int g, int h, int i) {
    int result;

    result = (f + g) - (h + i);
    return result;
}
```

ARM Assembly Code

```
; R4 = y
MAIN
. . .
MOV R0, #2      ; argument 0 = 2
MOV R1, #3      ; argument 1 = 3
MOV R2, #4      ; argument 2 = 4
MOV R3, #5      ; argument 3 = 5
BL DIFFOFSUMS   ; call function
MOV R4, R0      ; y = returned value
. . .

; R4 = result
DIFFOFSUMS
ADD R8, R0, R1   ; R8 = f + g
ADD R9, R2, R3   ; R9 = h + i
SUB R4, R8, R9   ; result = (f + g) - (h + i)
MOV R0, R4       ; put return value in R0
MOV PC, LR       ; return to caller
```

Remember that PC and LR are alternative names for R15 and R14, respectively. ARM is unusual in that PC is part of the register set, so a function return can be done with a MOV instruction. Many other instruction sets keep the PC in a special register and use a special return or jump instruction to return from functions.

These days, ARM compilers do a function return using BX LR. The BX branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set described in [Section 6.7.1](#). This chapter doesn't use the Thumb or BX instructions and thus sticks with the ARMv4 MOV PC, LR method.

We will see in Chapter 7 that treating the PC as an ordinary register complicates the implementation of the processor.

Code Example 6.21 has some subtle errors. Code Examples 6.22–6.25 show improved versions of the program.

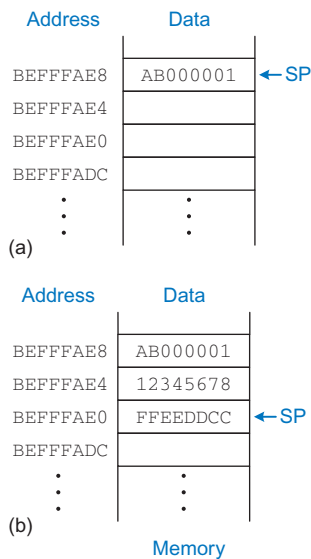


Figure 6.11 The stack (a) before expansion and (b) after two-word expansion

The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. ARM also allows for *ascending stacks* that grow up toward higher memory addresses. The stack pointer typically points to the topmost element on the stack; this is called a *full stack*. ARM also allows for *empty stacks* in which SP points one word beyond the top of the stack. The ARM *Application Binary Interface (ABI)* defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate. It specifies a *full descending stack*, which we will use in this chapter.

The Stack

The stack is memory that is used to save information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary values, we explain how the stack works.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be *popped* off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the ARM stack grows down in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.11 shows a picture of the stack. The stack pointer, SP (R13), is an ordinary ARM register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. SP points to (gives the address of) data. For example, in Figure 6.11(a), the stack pointer, SP, holds the address value 0xBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (SP) starts at a high memory address and decrements to expand as needed. Figure 6.11(b) shows the stack expanding to allow two more data words of temporary storage. To do so, SP decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing the return value. The `diffofsums` function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9. If `main` had been using these registers before the call to `diffofsums`, their contents would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

Code Example 6.22 shows an improved version of `diffofsums` that saves and restores `R4`, `R8`, and `R9`. Figure 6.12 shows the stack before, during, and after a call to the `diffofsums` function from Code Example 6.22. The stack starts at `0xBEF0F0FC`. `diffofsums` makes room for three words on the stack by decrementing the stack pointer `SP` by 12. It then stores the current values held in `R4`, `R8`, and `R9` in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, `R0` holds the result, but there



Code Example 6.22 FUNCTION SAVING REGISTERS ON THE STACK

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
    SUB SP, SP, #12      ; make space on stack for 3 registers
    STR R9, [SP, #8]     ; save R9 on stack
    STR R8, [SP, #4]     ; save R8 on stack
    STR R4, [SP]         ; save R4 on stack

    ADD R8, R0, R1       ; R8 = f + g
    ADD R9, R2, R3       ; R9 = h + i
    SUB R4, R8, R9       ; result = (f + g) - (h + i)
    MOV R0, R4           ; put return value in R0

    LDR R4, [SP]         ; restore R4 from stack
    LDR R8, [SP, #4]     ; restore R8 from stack
    LDR R9, [SP, #8]     ; restore R9 from stack
    ADD SP, SP, #12      ; deallocate stack space

    MOV PC, LR           ; return to caller
```

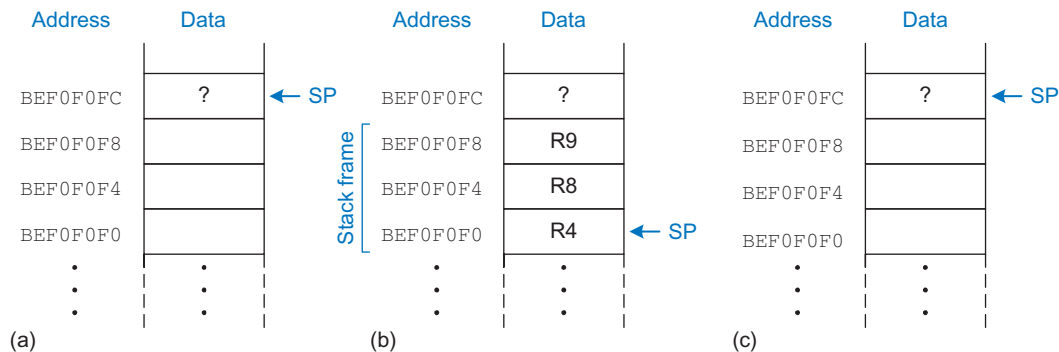


Figure 6.12 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

are no other side effects: R4, R8, R9, and SP have the same values as they did before the function call.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (LDM and STM) that are optimized to this purpose. Code Example 6.23 rewrites `diffofsums` using these instructions. The stack holds exactly the same information as in the previous example, but the code is much shorter.

Code Example 6.23 SAVING AND RESTORING MULTIPLE REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
    STMFD SP!, {R4, R8, R9}      ; push R4/8/9 on full descending stack

    ADD    R8, R0, R1            ; R8 = f + g
    ADD    R9, R2, R3            ; R9 = h + i
    SUB    R4, R8, R9            ; result = (f + g) - (h + i)
    MOV    R0, R4                ; put return value in R0

    LDMFD SP!, {R4, R8, R9}      ; pop R4/8/9 off full descending stack
    MOV    PC, LR                ; return to caller
```

LDM and STM come in four flavors for full and empty descending and ascending stacks (FD, ED, FA, EA). The `SP!` in the instructions indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load. `PUSH` and `POP` are synonyms for `STMFD SP!, {regs}` and `LDMFD SP!, {regs}`, respectively, and are the preferred way to save registers on the conventional full descending stack.

Preserved Registers

Code Examples 6.22 and 6.23 assume that all of the used registers (R4, R8, and R9) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, ARM divides registers into *preserved* and *nonpreserved* categories. The preserved registers include R4–R11. The nonpreserved registers are R0–R3 and R12. SP and LR (R13 and R14)

must also be preserved. A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.24 shows a further improved version of `diffofsums` that saves only R4 on the stack. It also illustrates the preferred `PUSH` and `POP` synonyms. The code reuses the nonpreserved argument registers R1 and R3 to hold the intermediate sums when those arguments are no longer necessary.

Code Example 6.24 REDUCING THE NUMBER OF PRESERVED REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
    PUSH {R4}           ; save R4 on stack
    ADD  R1, R0, R1      ; R1 = f + g
    ADD  R3, R2, R3      ; R3 = h + i
    SUB  R4, R1, R3      ; result = (f + g) - (h + i)
    MOV  R0, R4          ; put return value in R0
    POP  {R4}           ; pop R4 off stack
    MOV  PC, LR          ; return to caller
```

`PUSH` (and `POP`) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction. For example, `PUSH {R8, R1, R3}` will store R1 at the lowest memory address, then R3 and finally R8 at the next higher memory addresses on the stack.

Remember that when one function calls another, the former is the caller and the latter is the callee. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Table 6.6 summarizes which registers are preserved. R4–R11 are generally used to hold local variables within a function, so they must be saved. LR must also be saved, so that the function knows where to return.

Table 6.6 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

The convention of which registers are preserved or not preserved is part of the Procedure Call Standard for the ARM Architecture, rather than of the architecture itself. Alternate procedure call standards exist.

R0–R3 and R12 are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them.

R0–R3 are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. R0 certainly should not be preserved, because the callee returns its result in this register. Recall that the Current Program Status Register (CPSR) holds the condition flags. It is not preserved across function calls.

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above SP. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from SP at the beginning of the function.

The astute reader or an optimizing compiler may notice that the local variable `result` is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store it in the return register R0, eliminating the need to push and pop R4 and to move `result` from R4 to R0. Code Example 6.25 shows this even further optimized `diffofsums`.

Nonleaf Function Calls

A function that does not call others is called a *leaf function*; `diffofsums` is an example. A function that does call others is called a *nonleaf function*. As mentioned, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically:

Caller save rule: Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.

Code Example 6.25 OPTIMIZED `diffofsums` FUNCTION CALL

ARM Assembly Code

```
DIFFOFSUMS
ADD  R1, R0, R1    ; R1 = f + g
ADD  R3, R2, R3    ; R3 = h + i
SUB  R0, R1, R3    ; return (f + g) - (h + i)
MOV  PC, LR       ; return to caller
```

Code Example 6.26 demonstrates a nonleaf function `f1` and a leaf function `f2` including all the necessary saving and preserving of registers. Suppose `f1` keeps `i` in `R4` and `x` in `R5`. `f2` keeps `r` in `R4`. `f1` uses preserved registers `R4`, `R5`, and `LR`, so it initially pushes them on the stack according to the callee save rule. It uses `R12` to hold the intermediate result $(a - b)$ so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` pushes `R0` and `R1` onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. Although `R12` is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs `R12` and doesn't have to save it. `f1` then passes the argument to `f2` in `R0`, makes the function call, and uses the result in `R0`. `f1` then restores `R0` and `R1` because it still needs them. When `f1` is done, it puts the return value in `R0`, restores preserved registers `R4`, `R5`, and `LR`, and returns. `f2` saves and restores `R4` according to the callee save rule.

A nonleaf function overwrites `LR` when it calls another function using `BL`. Thus, a nonleaf function must always save `LR` on its stack and restore it before returning.

Code Example 6.26 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
    int i, x;

    x = (a + b) * (a - b);
    for (i = 0; i < a; i++)
        x = x + f2(b + i);
    return x;
}
```

```
int f2(int p) {
    int r;

    r = p + 5;
    return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
f1
    PUSH {R4, R5, LR}    ; save preserved registers used by f1
    ADD  R5, R0, R1      ; x = (a + b)
    SUB  R12, R0, R1     ; temp = (a - b)
    MUL  R5, R5, R12     ; x = x * temp = (a + b) * (a - b)
    MOV  R4, #0          ; i = 0
    FOR
        CMP  R4, R0      ; i < a?
        BGE  RETURN     ; no: exit loop
        PUSH {R0, R1}    ; save nonpreserved registers
        ADD  R0, R1, R4   ; argument is b + i
        BL   F2          ; call f2(b+i)
        ADD  R5, R5, R0   ; x = x + f2(b+i)
        POP  {R0, R1}    ; restore nonpreserved registers
        ADD  R4, R4, #1   ; i++
        B    FOR         ; continue for loop
    RETURN
    MOV  R0, R5          ; return value is x
    POP  {R4, R5, LR}    ; restore preserved registers
    MOV  PC, LR          ; return from f1
```

```
; R0 = p, R4 = r
f2
    PUSH {R4}            ; save preserved registers used by f2
    ADD  R4, R0, #5      ; r = p + 5
    ADD  R0, R4, R0      ; return value is r + p
    POP  {R4}            ; restore preserved registers
    MOV  PC, LR          ; return from f2
```

On careful inspection, one might note that `f2` does not modify `R1`, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call.

An optimizing compiler could observe that `f2` is a leaf procedure and could allocate `r` to a nonpreserved register, avoiding the need to save and restore `R4`.

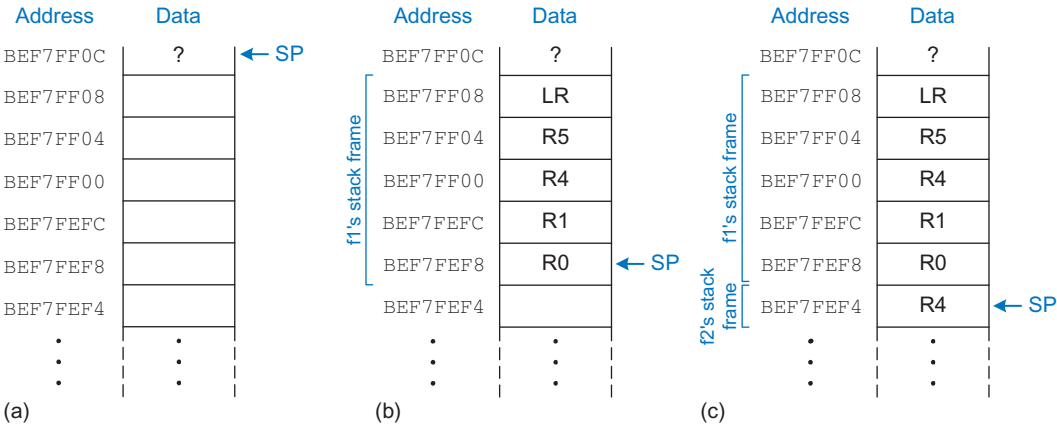


Figure 6.13 The stack: (a) before function calls, (b) during `f1`, and (c) during `f2`

Figure 6.13 shows the stack during execution of `f1`. The stack pointer originally starts at `0xBEF7FF0C`.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$. The factorial function can be rewritten recursively as $factorial(n) = n \times factorial(n - 1)$, as shown in Code Example 6.27. The factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address `0x8500`.

According to the callee save rule, `factorial` is a nonleaf function and must save LR. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save R0. Hence, it pushes both registers onto the stack at the start. It then checks whether $n \leq 1$. If so, it puts the return value of 1 in R0, restores the stack pointer, and returns to the caller. It does not have to reload LR and R0 in this case, because they were never modified. If $n > 1$, the function recursively calls `factorial(n - 1)`. It then restores the value of `n` and the link register (LR) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into R1, so as not to overwrite the returned value. The multiply instruction (`MUL R0, R1, R0`) multiplies `n` (R1) and the returned value (R0) and puts the result in R0.

Code Example 6.27 factorial RECURSIVE FUNCTION CALL

High-Level Code	ARM Assembly Code
<pre>int factorial(int n) { if (n <= 1) return 1; else return (n * factorial(n - 1)); }</pre>	<pre>0x8500 FACTORIAL PUSH {R0, LR} ; push n and LR on stack 0x8504 CMP R0, #1 ; R0 <= 1? 0x8508 BGT ELSE ; no: branch to else 0x850C MOV R0, #1 ; otherwise, return 1 0x8510 ADD SP, SP, #8 ; restore SP 0x8514 MOV PC, LR ; return 0x8518 ELSE SUB R0, R0, #1 ; n = n - 1 0x851C BL FACTORIAL ; recursive call 0x8520 POP {R1, LR} ; pop n (into R1) and LR 0x8524 MUL R0, R1, R0 ; R0 = n * factorial(n - 1) 0x8528 MOV PC, LR ; return</pre>

Figure 6.14 shows the stack when executing `factorial(3)`. For illustration, we show `SP` initially pointing to `0xBEFF0FF0`, as shown in Figure 6.14(a). The function creates a two-word stack frame to hold `n` (`R0`) and `LR`. On the first invocation, `factorial` saves `R0` (holding `n = 3`) at `0xBEFF0FE8` and `LR` at `0xBEFF0FEC`, as shown in Figure 6.14(b). The function then changes `n` to 2 and recursively calls `factorial(2)`, making `LR` hold `0x8520`. On the second invocation, it saves `R0` (holding `n=2`) at `0xBEFF0FE0` and `LR` at `0xBEFF0FE4`. This time, we know that `LR` contains `0x8520`. The function then changes `n` to 1 and recursively calls `factorial(1)`. On the third invocation, it saves `R0` (holding `n = 1`) at

For clarity, we will always save registers at the start of a procedure call. An optimizing compiler might observe that there is no need to save `R0` and `LR` when `n ≤ 1`, and thus push registers only in the `ELSE` portion of the function.

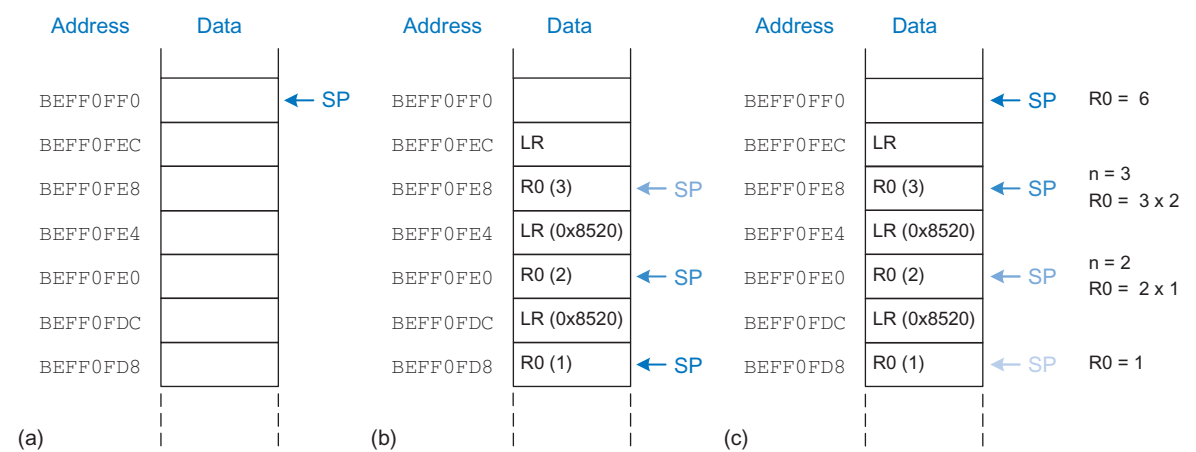


Figure 6.14 Stack: (a) before, (b) during, and (c) after factorial function call with `n = 3`

0xBEFF0FD8 and LR at 0xBEFF0FDC. This time, LR again contains 0x8520. The third invocation of `factorial` returns the value 1 in R0 and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` (into R1) to 2, restores LR to 0x8520 (it happened to already have this value), deallocates the stack frame, and returns $R0 = 2 \times 1 = 2$ to the first invocation. The first invocation restores `n` (into R1) to 3, restores LR to the return address of the caller, deallocates the stack frame, and returns $R0 = 3 \times 2 = 6$. Figure 6.14(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xBEFF0FF0), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. R0 holds the return value, 6.

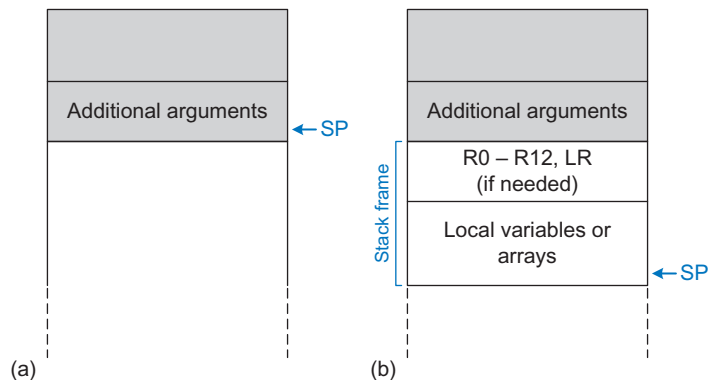
Additional Arguments and Local Variables*

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers. The stack is used to store this information. By ARM convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP. The caller must expand its stack to make room for the additional arguments. Figure 6.15(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in R4–R11; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.15(b) shows the organization of a callee's stack frame. The stack frame holds the temporary registers and link register (if they need to be saved because of a subsequent function call), and any of the saved

Figure 6.15 Stack usage: (a) before and (b) after call



registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*. This section describes ARM machine language and the tedious process of converting between assembly and machine language.

ARM uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would also encourage a single instruction format, but that is too restrictive. However, this issue allows us to introduce the last design principle:

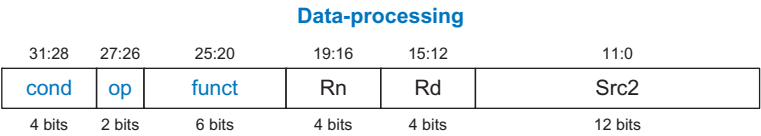
Design Principle 4: Good design demands good compromises.

ARM makes the compromise of defining three main instruction formats: *Data-processing*, *Memory*, and *Branch*. This small number of formats allows for some regularity among instructions, and thus simpler decoder hardware, while also accommodating different instruction needs. Data-processing instructions have a first source register, a second source that is either an immediate or a register, possibly shifted, and a destination register. The Data-processing format has several variations for these second sources. Memory instructions have three operands: a base register, an offset that is either an immediate or an optionally shifted register, and a register that is the destination on an LDR and another source on an STR. Branch instructions take one 24-bit immediate branch offset. This section discusses these ARM instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all the ARMv4 instructions.

6.4.1 Data-processing Instructions

The data-processing instruction format is the most common. The first source operand is a register. The second source operand can be an immediate or an optionally shifted register. A third register is the destination. [Figure 6.16](#) shows the data-processing instruction format. The 32-bit instruction has six fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*.

Figure 6.16 Data-processing instruction format



The operation the instruction performs is encoded in the fields highlighted in blue: *op* (also called the opcode or operation code) and *funct* or function code; the *cond* field encodes conditional execution based on flags described in Section 6.3.2. Recall that *cond* = 1110₂ for unconditional instructions. *op* is 00₂ for data-processing instructions.

The operands are encoded in the three fields: *Rn*, *Rd*, and *Src2*. *Rn* is the first source register and *Src2* is the second source; *Rd* is the destination register.

Figure 6.17 shows the format of the *funct* field and the three variations of *Src2* for data-processing instructions. *funct* has three subfields: *I*, *cmd*, and *S*. The *I*-bit is 1 when *Src2* is an immediate. The *S*-bit is 1 when the instruction sets the condition flags. For example, SUBS R1, R9, #11 has *S* = 1. *cmd* indicates the specific data-processing instruction, as given in Table B.1 in Appendix B. For example, *cmd* is 4 (0100₂) for ADD and 2 (0010₂) for SUB.

Three variations of *Src2* encoding allow the second source operand to be (1) an immediate, (2) a register (*Rm*) optionally shifted by a constant (*shamt5*), or (3) a register (*Rm*) shifted by another register (*Rs*). For the latter two encodings of *Src2*, *sh* encodes the type of shift to perform, as will be shown in Table 6.8.

Data-processing instructions have an unusual immediate representation involving an 8-bit unsigned immediate, *imm8*, and a 4-bit rotation, *rot*. *imm8* is rotated right by $2 \times rot$ to create a 32-bit constant. Table 6.7 gives example rotations and resulting 32-bit constants for the 8-bit immediate 0xFF. This representation is valuable because it

Rd is short for “register destination.” *Rn* and *Rm* unintuitively indicate the first and second register sources.

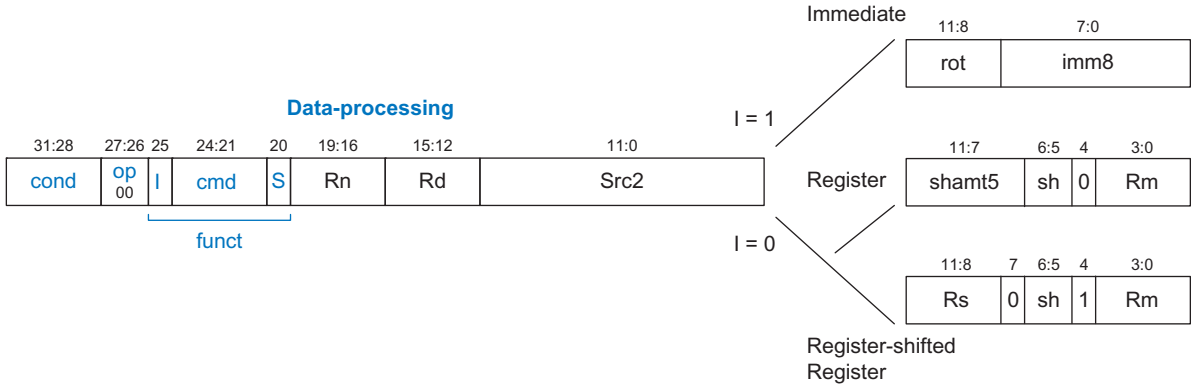


Figure 6.17 Data-processing instruction format showing the *funct* field and *Src2* variations

Table 6.7 Immediate rotations and resulting 32-bit constant for *imm8* = 0xFF

rot	32-bit Constant
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
...	...
1111	0000 0000 0000 0000 0000 0011 1111 1100

If an immediate has multiple possible encodings, the representation with the smallest rotation value *rot* is used. For example, #12 would be represented as (*rot*, *imm8*) = (0000, 00001100), not (0001, 00110000).

permits many useful constants, including small multiples of any power of two, to be packed into a small number of bits. Section 6.6.1 describes how to generate arbitrary 32-bit constants.

Figure 6.18 shows the machine code for ADD and SUB when *Src2* is a register. The easiest way to translate from assembly to machine code is to write out the values of each field and then convert these values to binary. Group the bits into blocks of four to convert to hexadecimal to make the machine language representation more compact. Beware that the destination is the first register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. *Rn* and *Rm* are the first and second source operands, respectively. For example, the assembly instruction ADD R5, R6, R7 has *Rn* = 6, *Rd* = 5, and *Rm* = 7.

Figure 6.19 shows the machine code for ADD and SUB with an immediate and two register operands. Again, the destination is the first

Assembly Code	Field Values											Machine Code										
	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
ADD R5, R6, R7 (0xE0865007)	1110 ₂	00 ₂	0	0100 ₂	0	6	5	0	0	0	7	1110	00	0	0100	0	0110	0101	00000	00	0	0111
SUB R8, R9, R10 (0xE049800A)	1110 ₂	00 ₂	0	0010 ₂	0	9	8	0	0	0	10	1110	00	0	0010	0	1001	1000	00000	00	0	1010
	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Figure 6.18 Data-processing instructions with three register operands

Assembly Code	Field Values										Machine Code									
	31:28	27:26	25	24:21	20	19:16	15:12	11:8		7:0	31:28	27:26	25	24:21	20	19:16	15:12	11:8		7:0
ADD R0, R1, #42 (0xE281002A)	1110 ₂	00 ₂	1	0100 ₂	0	1	0	0		42	1110	00	1	0100	0	0001	0000	0000		00101010
SUB R2, R3, #0xFF0 (0xE2432EFF)	1110 ₂	00 ₂	1	0010 ₂	0	3	2	14		255	1110	00	1	0010	0	0011	0010	1110		11111111
	cond	op	I	cmd	S	Rn	Rd	rot		imm8	cond	op	I	cmd	S	Rn	Rd	rot		imm8

Figure 6.19 Data-processing instructions with an immediate and two register operands

register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. The immediate of the ADD instruction (42) can be encoded in 8 bits, so no rotation is needed (*imm8* = 42, *rot* = 0). However, the immediate of SUB R2, R3, 0xFF0 cannot be encoded directly using the 8 bits of *imm8*. Instead, *imm8* is 255 (0xFF), and it is rotated right by 28 bits (*rot* = 14). This is easiest to interpret by remembering that the right rotation by 28 bits is equivalent to a left rotation by 32–28 = 4 bits.

Shifts are also data-processing instructions. Recall from Section 6.3.1 that the amount by which to shift can be encoded using either a 5-bit immediate or a register.

Figure 6.20 shows the machine code for logical shift left (LSL) and rotate right (ROR) with immediate shift amounts. The *cmd* field is 13 (1101₂) for all shift instruction, and the shift field (*sh*) encodes the type of shift to perform, as given in Table 6.8. *Rm* (i.e., R5) holds the 32-bit value to be shifted, and *shamt5* gives the number of bits to shift. The shifted result is placed in *Rd*. *Rn* is not used and should be 0.

Figure 6.21 shows the machine code for LSR and ASR with the shift amount encoded in the least significant 8 bits of *Rs* (R6 and R12). As

Table 6.8 *sh* field encodings

Instruction	sh	Operation
LSL	00 ₂	Logical shift left
LSR	01 ₂	Logical shift right
ASR	10 ₂	Arithmetic shift right
ROR	11 ₂	Rotate right

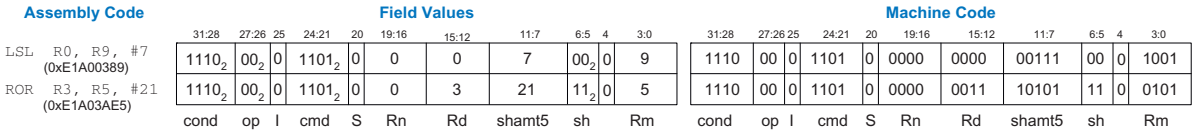


Figure 6.20 Shift instructions with immediate shift amounts

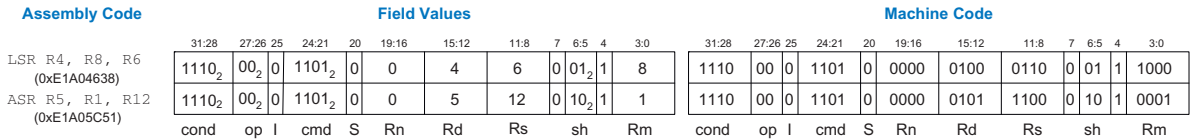


Figure 6.21 Shift instructions with register shift amounts

Table 6.9 Offset type control bits for memory instructions

Bit	\bar{I}	Meaning	U
0	Immediate offset in Src2	Subtract offset from base	
1	Register offset in Src2	Add offset to base	

before, *cmd* is 13 (1101₂), *sh* encodes the type of shift, *Rm* holds the value to be shifted, and the shifted result is placed in *Rd*. This instruction uses the *register-shifted register* addressing mode, where one register (*Rm*) is shifted by the amount held in a second register (*Rs*). Because the least significant 8 bits of *Rs* are used, *Rm* can be shifted by up to 255 positions. For example, if *Rs* holds the value 0xF001001C, the shift amount is 0x1C (28). A logical shift by more than 31 bits pushes all the bits off the end and produces all 0's. Rotate is cyclical, so a rotate by 50 bits is equivalent to a rotate by 18 bits.

6.4.2 Memory Instructions

Memory instructions use a format similar to that of data-processing instructions, with the same six overall fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*, as shown in Figure 6.22. However, memory instructions use a different *funct* field encoding, have two variations of *Src2*, and use an *op* of 01₂. *Rn* is the base register, *Src2* holds the offset, and *Rd* is the destination register in a load or the source register in a store. The offset is either a 12-bit unsigned immediate (*imm12*) or a register (*Rm*) that is optionally shifted by a constant (*shamt5*). *funct* is composed of six control bits: \bar{I} , *P*, *U*, *B*, *W*, and *L*. The \bar{I} (immediate) and *U* (add) bits determine whether the offset is an immediate or register and whether it should be added or subtracted, according to Table 6.9. The *P* (pre-index) and *W* (writeback) bits specify the index mode according to Table 6.10. The *L* (load) and *B* (byte) bits specify the type of memory operation according to Table 6.11.

Table 6.10 Index mode control bits for memory instructions

P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

Table 6.11 Memory operation type control bits for memory instructions

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

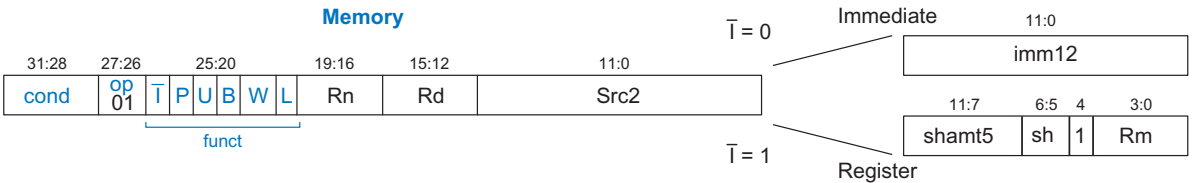


Figure 6.22 Memory instruction format for LDR, STR, LDRB, and STRB

Example 6.3 TRANSLATING MEMORY INSTRUCTIONS INTO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

STR R11, [R5], #-26

Notice the counterintuitive encoding of post-indexing mode.

Solution: STR is a memory instruction, so it has an *op* of 01₂. According to Table 6.11, *L* = 0 and *B* = 0 for STR. The instruction uses post-indexing, so according to Table 6.10, *P* = 0 and *W* = 0. The immediate offset is subtracted from the base, so \bar{I} = 0 and *U* = 0. Figure 6.23 shows each field and the machine code. Hence, the machine language instruction is 0xE405B01A.

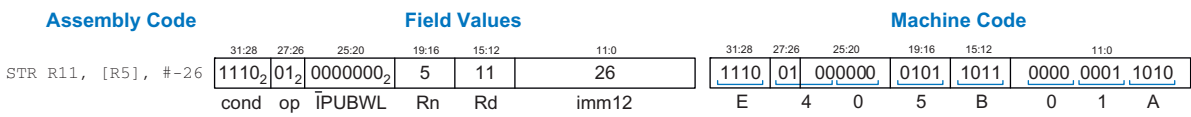


Figure 6.23 Machine code for the memory instruction of Example 6.3

6.4.3 Branch Instructions

Branch instructions use a single 24-bit signed immediate operand, *imm24*, as shown in Figure 6.24. As with data-processing and memory instructions, branch instructions begin with a 4-bit condition field and a 2-bit *op*, which is 10₂. The *funct* field is only 2 bits. The upper bit of *funct* is always 1 for branches. The lower bit, *L*, indicates the type of branch operation: 1 for BL and 0 for B. The remaining 24-bit two's complement *imm24* field is used to specify an instruction address relative to PC + 8.

Code Example 6.28 shows the use of the branch if less than (BLT) instruction and Figure 6.25 shows the machine code for that instruction. The *branch target address* (BTA) is the address of the next instruction to execute if the branch is taken. The BLT instruction in Figure 6.25 has a BTA of 0x80B4, the instruction address of the THERE label.

The 24-bit immediate field gives the number of instructions between the BTA and PC + 8 (two instructions past the branch). In this case, the value in the immediate field (*imm24*) of BLT is 3 because the BTA (0x80B4) is three instructions past PC + 8 (0x80A8).

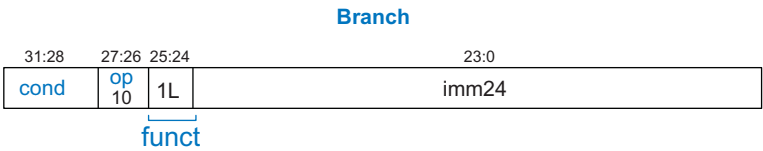


Figure 6.24 Branch instruction format

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

ARM Assembly Code	
0x80A0	BLT THERE
0x80A4	ADD R0, R1, R2
0x80A8	SUB R0, R0, R9
0x80AC	ADD SP, SP, #8
0x80B0	MOV PC, LR
0x80B4 THERE	SUB R0, R0, #1
0x80B8	ADD R3, R3, #0x5

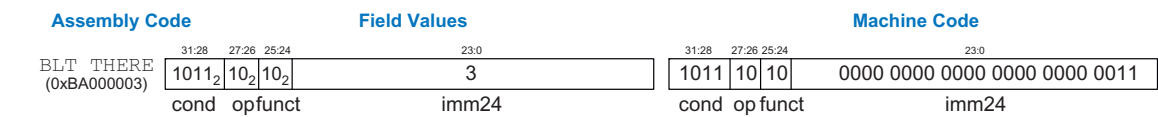


Figure 6.25 Machine code for branch if less than (BLT)

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC + 8.

Example 6.4 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

0x8040	TEST	LDRB	R5, [R0, R3]
0x8044		STRB	R5, [R1, R3]
0x8048		ADD	R3, R3, #1
0x8044		MOV	PC, LR
0x8050		BL	TEST
0x8054		LDR	R3, [R1], #4
0x8058		SUB	R4, R3, #9

Solution: Figure 6.26 shows the machine code for the branch and link instruction (BL). Its branch target address (0x8040) is six instructions behind PC + 8 (0x8058), so the immediate field is -6.

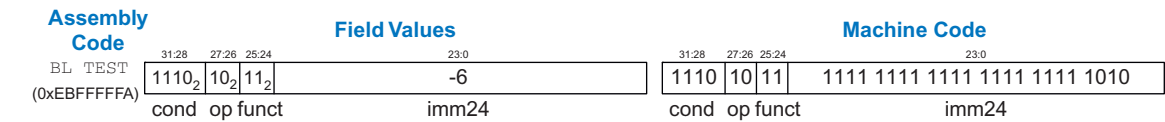


Figure 6.26 BL machine code

ARM is unusual among RISC architectures in that it allows the second source operand to be shifted in register and base addressing modes. This requires a shifter in series with the ALU in the hardware implementation but significantly reduces code length in common programs, especially array accesses. For example, in an array of 32-bit data elements, the array index must be left-shifted by 2 to compute the byte offset into the array. Any type of shift is permitted, but left shifts for multiplication are most common.

6.4.4 Addressing Modes

This section summarizes the modes used for addressing instruction operands. ARM uses four main modes: register, immediate, base, and PC-relative addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you easily learn other assembly languages. Register and base addressing have several submodes described below. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC). Table 6.12 summarizes and gives examples of each addressing mode.

Data-processing instructions use register or immediate addressing, in which the first source operand is a register and the second is a register or immediate, respectively. ARM allows the second register to be optionally shifted by an amount specified in an immediate or a third register. Memory instructions use base addressing, in which the base address comes from a register and the offset comes from an immediate, a register, or a register shifted by an immediate. Branches use PC-relative addressing in which the branch target address is computed by adding an offset to PC + 8.

6.4.5 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	ADD R3, R2, R1	$R3 \leftarrow R2 + R1$
Immediate-shifted register	SUB R4, R5, R9, LSR #2	$R4 \leftarrow R5 - (R9 \gg 2)$
Register-shifted register	ORR R0, R10, R2, ROR R7	$R0 \leftarrow R10 \mid (R2 \text{ ROR } R7)$
Immediate	SUB R3, R2, #25	$R3 \leftarrow R2 - 25$
Base		
Immediate offset	STR R6, [R11, #77]	$\text{mem}[R11+77] \leftarrow R6$
Register offset	LDR R12, [R1, -R5]	$R12 \leftarrow \text{mem}[R1 - R5]$
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	$R8 \leftarrow \text{mem}[R9 + (R2 \ll 2)]$
PC-Relative	B LABEL1	Branch to LABEL1

formats start with a 4-bit condition field and a 2-bit *op*. The best place to begin is to look at the *op*. If it is 00_2 , then the instruction is a data-processing instruction; if it is 01_2 , then the instruction is a memory instruction; if it is 10_2 , then it is a branch instruction. Based on that, the rest of the fields can be interpreted.

Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

0xE0475001
0xE5949010

Solution: First, we represent each instruction in binary and look at bits 27:26 to find the *op* for each instruction, as shown in Figure 6.27. The *op* fields are 00_2 and 01_2 , indicating a data-processing and memory instruction, respectively. Next, we look at the *funct* field of each instruction.

The *cmd* field of the data-processing instruction is 2 (0010_2) and the *I*-bit (bit 25) is 0, indicating that it is a SUB instruction with a register *Src2*. *Rd* is 5, *Rn* is 7, and *Rm* is 1.

The *funct* field for the memory instruction is 011001_2 . *B* = 0 and *L* = 1, so this is an LDR instruction. *P* = 1 and *W* = 0, indicating offset addressing. \bar{I} = 0, so the offset is an immediate. *U* = 1, so the offset is added. Thus, it is a load register instruction with an immediate offset that is added to the base register. *Rd* is 9, *Rn* is 4, and *imm12* is 16. Figure 6.27 shows the assembly code equivalent of the two machine instructions.

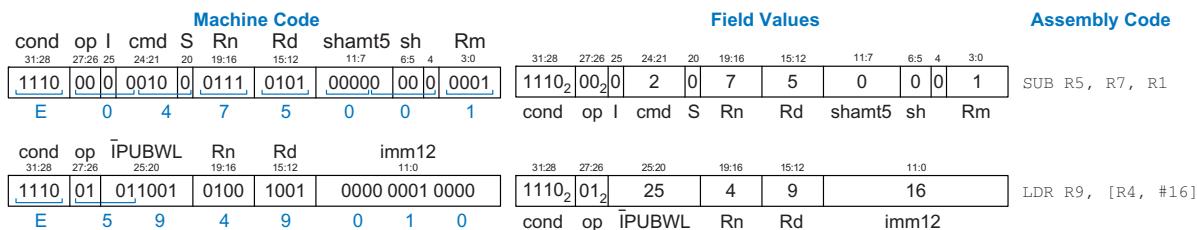


Figure 6.27 Machine code to assembly code translation

6.4.6 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different

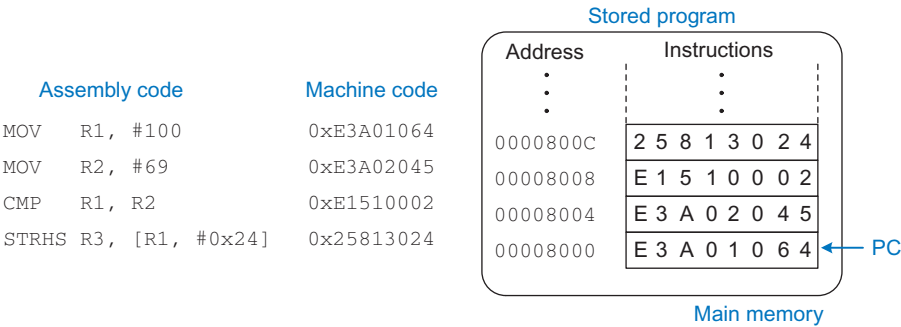


Figure 6.28 Stored program

program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory reads and instruction executions.

Figure 6.28 shows how machine instructions are stored in memory. In ARM programs, the instructions are normally stored starting at low addresses, in this case 0x00008000. Remember that ARM memory is byte-addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC), which is register R15. For historical reasons, a read to the PC returns the address of the current instruction plus 8.

To execute the code in Figure 6.28, the PC is initialized to address 0x00008000. The processor fetches the instruction at that memory address and executes the instruction, 0xE3A01064 (MOV R1, #100). The processor then increments the PC by 4 to 0x00008004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For ARM, the architectural state includes the register file and status registers. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

