1.1 Welcome to Assembly Language

Assembly Language for Intel-Based Computers focuses on programming microprocessors compatible with the Intel IA-32 processor family on the MS-Windows platform. You can use an Intel or AMD 32-bit/64-bit processor to run all programs in this book.

The IA-32 family began with the Intel 80386, continuing to (and including) the Pentium 4. Microsoft MASM (Macro Assembler) 8.0 is our assembler of choice, running under MS-Windows. There are other good assemblers for Intel-based computers, including TASM (Turbo Assembler), NASM (Netwide Assembler), and the GNU assembler. Of these, TASM has the most similar syntax to MASM, and you could (with some help from your instructor) assemble and run most of the programs in this book. The other assemblers, NASM and GNU, have a somewhat different syntax.

Assembly language is the oldest programming language, and of all languages, bears the closest resemblance to native machine language. It provides direct access to computer hardware, requiring you to understand much about your computer’s architecture and operating system.
Educational Value  Why read this book? Perhaps you’re taking a college course whose name is similar to one of these:

• Microcomputer Assembly Language
• Assembly Language Programming
• Introduction to Computer Architecture
• Fundamentals of Computer Systems
• Embedded Systems Programming

These are names of courses at colleges and universities using previous editions of this book. This book covers basic principles about computer architecture, machine language, and low-level programming. You will learn enough assembly language to test your knowledge on today’s most widely used microprocessor family. You won’t be learning to program a “toy” computer using a simulated assembler; MASM is an industrial-strength assembler, used by practicing professionals. You will learn the architecture of the Intel IA-32 processor family from a programmer’s point of view.

If you doubt the value of low-level programming and studying details of computer software and hardware, take note of the following quote from a leading computer scientist, Donald Knuth, in discussing his famous book series, The Art of Computer Programming:

Some people [say] that having machine language, at all, was the great mistake that I made. I really don’t think you can write a book for serious computer programmers unless you are able to discuss low-level detail.1

Visit this book’s Web site to get lots of supplemental information, tutorials, and exercises at www.asmirvine.com

1.1.1 Good Questions to Ask

What Background Should I Have?  Before reading this book, you should have completed a college-level introductory computer programming course. You will better understand high-level programming constructs such as IF statements, loops, and arrays when implemented in assembly language.

What Are Assemblers and Linkers?  An assembler is a utility program that converts source code programs from assembly language into machine language. A linker is a utility program that combines individual files created by an assembler into a single executable program. A related utility, called a debugger, lets you to step through a program while it’s running and examine registers and memory.

What Hardware and Software Do I Need?  You need a computer with Intel386, Intel486, Pentium, or compatible processor. AMD processors, for example, work very well with this book. MASM (the assembler) is compatible with all 32-bit versions of Microsoft Windows, beginning with Windows 95. A few of the advanced programs relating to direct hardware access and disk sector programming must be run under MS-DOS, Windows 95/98/Me, because of tight security restrictions imposed by Windows NT/2000/XP.

In addition, you will need the following:

• Editor: Use a text editor or programmer’s editor to create assembly language source files. The CD-ROM accompanying this book contains Microsoft Visual C++ 2005 Express. It has an excellent text editor in its integrated development environment.
• 32-Bit Debugger: Strictly speaking, you don’t need a debugger, but you will probably want one. The debugger supplied with Visual C++ 2005 Express is excellent.
What Types of Programs Will I Create? This book shows how to create two general classes of programs:

- **16-Bit Real-Address Mode:** 16-bit real-address mode programs run under MS-DOS and in the console window under MS-Windows. Also known as *real mode* programs, they use a segmented memory model required of programs written for the Intel 8086 and 8088 processors. There are notes throughout the book with tips about programming in real-address mode, and two chapters are exclusively devoted to color and graphics programming in real mode.

- **32-Bit Protected Mode:** 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real mode programs.

What Do I Get with This Book? Besides a lot of printed paper, you get a CD-ROM attached to the book containing Visual C++ 2005 Express. You will be able to download the Microsoft Assembler from the Microsoft Web site. See www.asmirvine.com for details on how to obtain the assembler.

The book’s Web site (www.asmirvine.com) has the following:

- **Online Help File** detailing the book’s library procedures and essential Windows API structures, by Gerald Cahill.
- **Assembly Language Workbook**, a collection of tutorials by the author.
- **Irvine32 and Irvine16 link libraries** for real-address mode and protected mode programming, with complete source code.
- **Example programs** with all source code from the book.
- **Corrections** to the book and example programs. Hopefully not too many!
- **Tutorials** on installing the assembler.
- **Articles** on advanced topics not included in the printed book for lack of space.
- **Discussion Group**, which over 500 members have joined.

What Will I Learn? This book should make you better informed about computer architecture, programming, and computer science. Here’s what you will learn:

- Basic principles of computer architecture as applied to the Intel IA-32 processor family.
- Basic boolean logic and how it applies to programming and computer hardware.
- How IA-32 processors manage memory, using real mode, protected mode, and virtual mode.
- How high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code.
- How high-level languages implement arithmetic expressions, loops, and logical structures at the machine level.
- Data representation, including signed and unsigned integers, real numbers, and character data.
- How to debug programs at the machine level. The need for this skill is vital when you work in languages such as C and C++, which provide access to low-level data and hardware.
- How application programs communicate with the computer’s operating system via interrupt handlers, system calls, and common memory areas.
- How to interface assembly language code to C++ programs.
- How to create assembly language application programs.

How Does Assembly Language Relate to Machine Language? *Machine language* is a numeric language specifically understood by a computer’s processor (the CPU). IA-32–compatible
processors understand a common machine language. Assembly language consists of statements
written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a one-to-one
relationship with machine language: Each assembly language instruction corresponds to a
single machine-language instruction.

How Do C++ and Java Relate to Assembly Language? High-level languages such as C++ and
Java have a one-to-many relationship with assembly language and machine language. A single state-
ment in C++ expands into multiple assembly language or machine instructions. We can show how
C++ statements expand into machine code. Most people cannot read raw machine code, so we will
use its closest relative, assembly language. The following C++ statement carries out two arithmetic
operations and assigns the result to a variable. Assume X and Y are integers:

```cpp
int Y;
int X = (Y + 4) * 3;
```

Following is the statement’s translation to assembly language. The translation requires multiple
statements because assembly language works at a detailed level:

```assembly
mov eax,Y ; move Y to the EAX register
add eax,4 ; add 4 to the EAX register
mov ebx,3 ; move 3 to the EBX register
imul ebx ; multiply EAX by EBX
mov X,eax ; move EAX to X
```

(Registers are named storage locations in the CPU that hold intermediate results of operations.)

The point in this example is not to claim that C++ is superior to assembly language or vice versa,
but to show their relationship.

Is Assembly Language Portable? An important distinction between high-level languages and
assembly language has to do with portability. A language whose source programs can be compiled
and run on a wide variety of computer systems is said to be portable. A C++ program, for example,
should compile and run on just about any computer, unless it makes specific references to library
functions existing under a single operating system. A major feature of the Java language is that
compiled programs run on nearly any computer system.

Assembly language is not portable because it is designed for a specific processor family.
There are a number of different assembly languages widely used today, each based on a processor
family. Some well-known processor families are Motorola 68x00, Intel IA-32, SUN Sparc,
Vax, and IBM-370. The instructions in assembly language may directly match the computer’s
architecture or they may be translated during execution by a program inside the processor known
as a microcode interpreter.

Why Learn Assembly Language? Why not just read a good book on computer hardware and
architecture and avoid learning assembly language programming?

- If you study computer engineering, you may likely be asked to write embedded programs. They
  are short programs stored in a small amount of memory in single-purpose devices such as
telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, data acquisition instruments, video cards, sound cards, hard drives, modems, and printers. Assembly language is ideal for writing embedded programs because of its economical use of memory.

- Real-time applications such as simulations and hardware monitoring require precise timing and responses. High-level languages do not give programmers exact control over machine code generated by compilers. Assembly language permits you to precisely specify a program’s executable code.
- Computer game consoles require their software to be highly optimized for small code size and fast execution. Game programmers are experts at writing code that takes full advantage of hardware features in a target system. They use assembly language as their tool of choice because it permits direct access to computer hardware, and code can be hand optimized for speed.
- Assembly language helps you to gain an overall understanding of the interaction between computer hardware, operating systems, and application programs. Using assembly language, you can apply and test theoretical information you are given in computer architecture and operating systems courses.
- Application programmers occasionally find that limitations in high-level languages prevent them from efficiently performing low-level tasks such as bitwise manipulation and data encryption. They will often call subroutines written in assembly language to accomplish their goal.
- Hardware manufacturers create device drivers for the equipment they sell. Device drivers are programs that translate general operating system commands into specific references to hardware details. Printer manufacturers, for example, create a different MS-Windows device driver for each model they sell. The same is true for Mac OS, Linux, and other operating systems.

Are There Rules in Assembly Language? Most rules in assembly language are based on physical limitations of the target processor and its machine language. The CPU, for example, requires two instruction operands to be the same size. Assembly language has fewer rules than C++ or Java because the latter use syntax rules to reduce unintended logic errors at the expense of low-level data access. Assembly language programmers can easily bypass restrictions characteristic of high-level languages. Java, for example, does not permit access to specific memory addresses. One can work around the restriction by calling a C subroutine using JNI (Java Native Interface) classes, but the resulting program can be awkward to maintain. Assembly language, on the other hand, can access any memory address. The price for such freedom is high: Assembly language programmers spend a lot of time debugging!

1.1.2 Assembly Language Applications

In the early days of programming, most applications were written partially or entirely in assembly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. Programmers switched to high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability. More recently, object-oriented languages such as C++, C#, and Java have made it possible to write complex programs containing millions of lines of code.

It is rare to see large application programs coded completely in assembly language because they would take too much time to write and maintain. Instead, assembly language is used to optimize certain sections of application programs for speed and to access computer hardware. Table 1-1 compares the adaptability of assembly language to high-level languages in relation to various types of applications.
C++ has the unique quality of offering a compromise between high-level structure and low-level details. Direct hardware access is possible but completely nonportable. Most C++ compilers have the ability to generate assembly language source code, which the programmer can customize and refine before assembling into executable code.

### 1.1.3 Section Review

1. How do assemblers and linkers work together?
2. How will studying assembly language enhance your understanding of operating systems?
3. What is meant by a *one-to-many relationship* when comparing a high-level language to machine language?
4. Explain the concept of *portability* as it applies to programming languages.
5. Is the assembly language for the Intel 80x86 processor family the same as those for computer systems such as the Vax or Motorola 68x00?
6. Give an example of an *embedded systems* application.
7. What is a device driver?
8. Do you suppose type checking on pointer variables is stronger (stricter) in assembly language or in C++?
9. Name two types of applications that would be better suited to assembly language than a high-level language.
10. Why would a high-level language not be an ideal tool for writing a program to directly access a particular brand of printer?
11. Why is assembly language not usually used when writing large application programs?
12. **Challenge:** Translate the following C++ expression to assembly language, using the example presented earlier in this chapter as a guide: \( X = (Y \times 4) + 3. \)
1.2 Virtual Machine Concept

A most effective way to explain how a computer’s hardware and software are related is called the virtual machine concept. Our explanation of this model is derived from Andrew Tanenbaum’s book, *Structured Computer Organization*. To explain this concept, let us begin with the most basic function of a computer, executing programs.

A computer can usually execute programs written in its native machine language. Each instruction in this language is simple enough to be executed using a relatively small number of electronic circuits. For simplicity, we will call this language L0.

Programmers would have a difficult time writing programs in L0 because it is enormously detailed and consists purely of numbers. If a new language, L1, could be constructed that was easier to use, programs could be written in L1. There are two ways to achieve this:

- **Interpretation:** As the L1 program is running, each of its instructions could be decoded and executed by a program written in language L0. The L1 program begins running immediately, but each instruction has to be decoded before it can execute.

- **Translation:** The entire L1 program could be converted into an L0 program by an L0 program specifically designed for this purpose. Then the resulting L0 program could be executed directly on the computer hardware.

**Virtual Machines** Rather than using only languages, it is easier to think in terms of a hypothetical computer, or virtual machine, at each level. The virtual machine VM1, as we will call it, can execute commands written in language L1. The virtual machine VM0 can execute commands written in language L0:

Each virtual machine can be constructed of either hardware or software. People can write programs for virtual machine VM1, and if it is practical to implement VM1 as an actual computer, programs can be executed directly on the hardware. Or programs written in VM1 can be interpreted/translated and executed on machine VM0.

Machine VM1 cannot be radically different from VM0 because the translation or interpretation would be too time-consuming. What if the language VM1 supports is still not programmer-friendly enough to be used for useful applications? Then another virtual machine, VM2, can be designed that is more easily understood. This process can be repeated until a virtual machine VMn can be designed to support a powerful, easy-to-use language.

The Java programming language is based on the virtual machine concept. A program written in the Java language is translated by a Java compiler into Java byte code. The latter is a low-level language quickly executed at run time by a program known as a Java virtual machine (JVM). The JVM has been implemented on many different computer systems, making Java programs relatively system independent.

**Specific Machines** Let us relate this to actual computers and languages, using names such as Level 1 for VM1 and Level 0 for VM0, shown in Figure 1–1. A computer’s digital logic hardware represents machine Level 0, and Level 1 is implemented by an interpreter hardwired into the processor called microarchitecture. Above this is Level 2, called the instruction set architecture. This is the first level at which users can typically write programs, although the programs consist of binary numbers.
Microarchitecture (Level 1) Computer chip manufacturers don’t generally make it possible for average users to write microinstructions. The specific microarchitecture commands are often a proprietary secret. It might require three or four microcode instructions to carry out a primitive operation such as fetching a number from memory and incrementing it by 1.

Figure 1–1 Virtual Machine Levels 0 through 5.

Instruction Set Architecture (Level 2) Computer chip manufacturers design into the processor an instruction set to carry out basic operations, such as move, add, or multiply. This set of instructions is also referred to as conventional machine language, or simply machine language. Each machine-language instruction is executed by several microinstructions.

Operating System (Level 3) As computers evolved, additional virtual machines were created to enable programmers to be more productive. A Level 3 machine understands interactive commands by users to load and execute programs, display directories, and so forth. This is known as the computer’s operating system. The operating system software is translated into machine code running on a Level 2 machine.

Assembly Language (Level 4) Above the operating system level, programming languages provide the translation layers to make large-scale software development practical. Assembly language, which appears at Level 4, uses short mnemonics such as ADD, SUB, and MOV, which are easily translated to the instruction set architecture level (Level 2). Other assembly language statements, such as Interrupt calls, are executed directly by the operating system (Level 3). Assembly language programs are translated (assembled) in their entirety into machine language before they begin to execute.

High-Level Languages (Level 5) At Level 5 are high-level programming languages such as C++, C#, Visual Basic, and Java. Programs in these languages contain powerful statements that translate into multiple instructions at Level 4. Internally, compilers translate Level 5 programs into Level 4 programs, which are in turn translated into Level 4 code. The latter are assembled into conventional machine language.
1.2.2 Section Review
1. In your own words, describe the virtual machine concept.
2. Why don’t programmers write application programs in machine language?
3. (True/False): When an interpreted program written in language L1 runs, each of its instructions is decoded and executed by a program written in language L0.
4. Explain the technique of translation when dealing with languages at different virtual machine levels.
5. How does the Intel IA-32 processor architecture contain an example of a virtual machine?
6. What software permits compiled Java programs to run on almost any computer?
7. Name the six virtual machine levels named in this section, from lowest to highest.
8. Why don’t programmers write applications in microcode?
9. Conventional machine language is used at which level of the virtual machine shown in Figure 1–1?
10. Statements at the assembly language level of a virtual machine are translated into statements at which other level(s)?

1.3 Data Representation
Before discussing computer organization and assembly language, let us clarify binary, hexadecimal, decimal, and character-based storage concepts. Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers. Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used. Programmers develop a certain fluency with number formats and can quickly translate numbers from one format to another.

Each numbering format, or system, has a base, or maximum number of symbols that can be assigned to a single digit. Table 1-2 shows the possible digits for the numbering systems used most commonly in computer literature. In the last row of the table, hexadecimal numbers use the digits 0 through 9 and continue with the letters A through F to represent decimal values 10 through 15. It is quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.
1.3.1 Binary Numbers

A computer stores instructions and data in memory as collections of electronic charges. Representing these entities with numbers requires a system geared to the concepts of on and off or true and false. Binary numbers are base 2 numbers in which each binary digit (called a bit) is either a 0 or a 1. Bits are numbered starting at zero on the right side and increasing toward the left. The bit on the left is called the most significant bit (MSB), and the bit on the right is the least significant bit (LSB). The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:

```
MSB        LSB
1 0 1 1 0 0 1 0 1 0 1 1 1 0 0
15 0
```

Binary integers can be signed or unsigned. A signed integer is positive or negative. An unsigned integer is by default positive. Zero is considered positive. Using special encoding schemes, one can represent real numbers in binary, but we will defer the discussion of these for a later chapter. For now, let’s begin with unsigned binary integers.

**Unsigned Binary Integers**

Starting with the least significant bit, each bit in an unsigned binary integer represents an increasing power of 2. The following figure contains an 8-bit binary number, showing how powers of two increase from right to left:

```
1 1 1 1 1 1 1 1
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
```

Table 1-3 lists the decimal values of $2^0$ through $2^{15}$.

<table>
<thead>
<tr>
<th>$2^n$</th>
<th>Decimal Value</th>
<th>$2^n$</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>1</td>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2</td>
<td>$2^9$</td>
<td>512</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4</td>
<td>$2^{10}$</td>
<td>1024</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8</td>
<td>$2^{11}$</td>
<td>2048</td>
</tr>
<tr>
<td>$2^4$</td>
<td>16</td>
<td>$2^{12}$</td>
<td>4096</td>
</tr>
<tr>
<td>$2^5$</td>
<td>32</td>
<td>$2^{13}$</td>
<td>8192</td>
</tr>
<tr>
<td>$2^6$</td>
<td>64</td>
<td>$2^{14}$</td>
<td>16384</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
<td>$2^{15}$</td>
<td>32768</td>
</tr>
</tbody>
</table>
**Translating Unsigned Binary Integers to Decimal**

*Weighted positional notation* represents a convenient way to calculate the decimal value of an unsigned binary integer having \( n \) digits:

\[
\text{dec} = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \cdots + (D_1 \times 2^1) + (D_0 \times 2^0)
\]

\( D \) indicates a binary digit. For example, binary 00001001 is equal to 9. We calculate this value by leaving out terms equal to zero:

\[
(1 \times 2^3) + (1 \times 2^0) = 9
\]

The same calculation is shown by the following figure:

```
     + 8
  +---+---+---+---+---+---+
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  |   |   |   |   |   |   |
  +---+---+---+---+---+---+
```

```
0 0 0 0 1 0 0 1
```

**Translating Unsigned Decimal Integers to Binary**

To translate an unsigned decimal integer into binary, repeatedly divide the integer by 2, saving each remainder as a binary digit. The following table shows the steps required to translate decimal 37 to binary. The remainder digits, starting from the top row, are the binary digits \( D_5, D_4, D_3, D_2, D_1, \) and \( D_0 \):

<table>
<thead>
<tr>
<th>Division</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 / 2</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>18 / 2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>9 / 2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4 / 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2 / 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 / 2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Collecting the binary digits in the remainder column in reverse order produces binary 100101. Because Intel computer storage always consists of binary numbers whose lengths are multiples of 8, we fill the remaining two digit positions on the left with zeros, producing 00100101.

### 1.3.2 Binary Addition

When adding two binary integers, proceed bit by bit, starting with the low-order pair of bits (on the right) and add each subsequent pair of bits. There are four ways to add two binary digits, as shown here:

<table>
<thead>
<tr>
<th>( 0 + 0 )</th>
<th>( 0 + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

When adding 1 to 1, the result is 10 binary (think of it as the decimal value 2). The extra
digit generates a carry to the next-highest bit position. In the following figure, we add binary 00000100 to 00000111:

\[
\begin{array}{c}
\text{carry: 1} \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
+ & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

Beginning with the lowest bit in each number (bit position 0), we add 0 + 1, producing a 1 in the bottom row. The same happens in the next highest bit (position 1). In bit position 2, we add 1 + 1, generating a sum of zero and a carry of 1. In bit position 3, we add the carry bit to 0 + 0, producing 1. The rest of the bits are zeros. You can verify the addition by adding the decimal equivalents shown on the right side of the figure (4 + 7 = 11).

### 1.3.3 Integer Storage Sizes

The basic storage unit for all data in an IA-32–based computer is a byte, containing 8 bits. Other storage sizes are word (2 bytes), doubleword (4 bytes), and quadword (8 bytes). In the following figure, the number of bits is shown for each size:

- byte: 8
- word: 16
- doubleword: 32
- quadword: 64

Table 1-4 shows the range of possible values for each type of unsigned integer.

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Range (Low–High)</th>
<th>Powers of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned byte</td>
<td>0 to 255</td>
<td>0 to (2^8 - 1)</td>
</tr>
<tr>
<td>Unsigned word</td>
<td>0 to 65,535</td>
<td>0 to (2^{16} - 1)</td>
</tr>
<tr>
<td>Unsigned doubleword</td>
<td>0 to 4,294,967,295</td>
<td>0 to (2^{32} - 1)</td>
</tr>
<tr>
<td>Unsigned quadword</td>
<td>0 to 18,446,744,073,709,551,615</td>
<td>0 to (2^{64} - 1)</td>
</tr>
</tbody>
</table>

**Large Measurements** A number of large measurements are used when referring to both memory and disk space:

- One kilobyte is equal to \(2^{10}\), or 1024 bytes.
- One megabyte (MB) is equal to \(2^{20}\), or 1,048,576 bytes.
- One gigabyte (GB) is equal to \(2^{30}\), or 1024\(^3\), or 1,073,741,824 bytes.
1.3 Data Representation

• One terabyte (TB) is equal to \(2^{40}\), or 1024\(^4\), or 1,099,511,627,776 bytes.
• One petabyte is equal to \(2^{50}\), or 1,125,899,906,842,624 bytes.
• One exabyte is equal to \(2^{60}\), or 1,152,921,504,606,846,976 bytes.
• One zettabyte is equal to \(2^{70}\) bytes.
• One yottabyte is equal to \(2^{80}\) bytes.

1.3.4 Hexadecimal Integers

Large binary numbers are cumbersome to read, so hexadecimal digits offer a convenient way to represent binary data. Each digit in a hexadecimal integer represents four binary bits, and two hexadecimal digits together represent a byte. A single hexadecimal digit represents decimal 0 to 15, so letters A to F represent decimal values in the range 10 through 15. Table 1-5 shows how each sequence of four binary bits translates into a decimal or hexadecimal value.

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>

The following example shows how binary 0001011101010011110010100 is equivalent to hexadecimal 16A794:

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>A</th>
<th>7</th>
<th>9</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0110</td>
<td>1010</td>
<td>0111</td>
<td>1001</td>
<td>0100</td>
</tr>
</tbody>
</table>

Converting Unsigned Hexadecimal to Decimal

In hexadecimal, each digit position represents a power of 16. This is helpful when calculating the decimal value of a hexadecimal integer. Suppose we number the digits in a four-digit hexadecimal integer with subscripts as \(D_3D_2D_1D_0\). The following formula calculates the number’s decimal value:

\[
\text{dec} = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)
\]

The formula can be generalized for any \(n\)-digit hexadecimal number:

\[
\text{dec} = (D_{n-1} \times 16^{n-1}) + (D_{n-2} \times 16^{n-2}) + \cdots + (D_1 \times 16^1) + (D_0 \times 16^0)
\]
For example, hexadecimal 1234 is equal to \((1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)\), or decimal 4660. Similarly, hexadecimal 3BA4 is equal to \((3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)\), or decimal 15,268. The following figure shows this last calculation:

Table 1-6 lists the powers of 16 from \(16^0\) to \(16^7\).

<table>
<thead>
<tr>
<th>(16^n)</th>
<th>Decimal Value</th>
<th>(16^n)</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(16^0)</td>
<td>1</td>
<td>(16^4)</td>
<td>65,536</td>
</tr>
<tr>
<td>(16^1)</td>
<td>16</td>
<td>(16^5)</td>
<td>1,048,576</td>
</tr>
<tr>
<td>(16^2)</td>
<td>256</td>
<td>(16^6)</td>
<td>16,777,216</td>
</tr>
<tr>
<td>(16^3)</td>
<td>4096</td>
<td>(16^7)</td>
<td>268,435,456</td>
</tr>
</tbody>
</table>

Converting Unsigned Decimal to Hexadecimal

To convert an unsigned decimal integer to hexadecimal, repeatedly divide the decimal value by 16 and retain each remainder as a hexadecimal digit. For example, the following table lists the steps when converting decimal 422 to hexadecimal:

<table>
<thead>
<tr>
<th>Division</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>422 / 16</td>
<td>26</td>
<td>6</td>
</tr>
<tr>
<td>26 / 16</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1 / 16</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

When one collects the digits from the remainder column in reverse order, the hexadecimal representation is \(1A6\). The same algorithm was used for binary numbers in Section 1.3.1.

1.3.5 Signed Integers

Signed binary integers are positive or negative. On Intel-based computers, the most significant bit (MSB) indicates the sign: 0 is positive and 1 is negative. The following figure shows examples of 8-bit negative and positive integers:
Two’s-Complement Notation

Negative integers use two's-complement representation, using the mathematical principle that the two’s complement of an integer is its additive inverse. (If you add a number to its additive inverse, the sum is zero.)

Two’s-complement representation is useful to processor designers because it removes the need for separate digital circuits to handle both addition and subtraction. For example, if presented with the expression $A - B$, the processor can simply convert it to an addition expression: $A + (-B)$.

The two’s complement of a binary integer is formed by inverting (complementing) its bits and adding 1. Using the 8-bit binary value 00000001, for example, its two’s complement turns out to be 11111111, as can be seen as follows:

<table>
<thead>
<tr>
<th>Starting value</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Reverse the bits</td>
<td>11111110</td>
</tr>
<tr>
<td>Step 2: Add 1 to the value from Step 1</td>
<td>11111110 + 00000001</td>
</tr>
<tr>
<td>Sum: Two’s-complement representation</td>
<td>11111111</td>
</tr>
</tbody>
</table>

11111111 is the two’s-complement representation of $-1$. The two’s-complement operation is reversible, so the two’s complement of 11111111 is 00000001.

Two’s Complement of Hexadecimal

To form the two’s complement of a hexadecimal integer, reverse all bits and add 1. An easy way to reverse the bits of a hexadecimal digit is to subtract the digit from 15. Here are several examples of hexadecimal integers converted to their two’s complements:

- $6A3D \rightarrow 95C2 + 1 \rightarrow 95C3$
- $95C3 \rightarrow 6A3C + 1 \rightarrow 6A3D$
- $21F0 \rightarrow DE0F + 1 \rightarrow DE10$
- $DE10 \rightarrow 21EF + 1 \rightarrow 21F0$

Converting Signed Binary to Decimal

To calculate the decimal equivalent of a signed binary integer, do one of the following:

- If the highest bit is a 1, the number is stored in two’s-complement notation. Form its two’s complement a second time to get its positive equivalent. Then convert this new number to decimal as if it were an unsigned binary integer.
- If the highest bit is a 0, you can convert it to decimal as if it were an unsigned binary integer.

For example, signed binary 11110000 has a 1 in the highest bit, indicating that it is a negative integer. First we form its two’s complement, then we convert the result to decimal. Here are the steps in the process:

<table>
<thead>
<tr>
<th>Starting value</th>
<th>11110000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1: Reverse the bits</td>
<td>00001111</td>
</tr>
<tr>
<td>Step 2: Add 1 to the value from Step 1</td>
<td>00001111 + 1</td>
</tr>
<tr>
<td>Step 3: Form the two’s complement</td>
<td>00010000</td>
</tr>
<tr>
<td>Step 4: Convert to decimal</td>
<td>16</td>
</tr>
</tbody>
</table>

Because the original integer (11110000) was negative, we infer its decimal value was $-16$. 
Converting Signed Decimal to Binary

To determine the binary representation of a signed decimal integer, do the following:

1. Convert the absolute value of the decimal integer to binary.
2. If the original decimal integer was negative, form the two’s complement of the binary number from the previous step.

For example, \(-43\) decimal is translated to binary as follows:

1. The binary representation of unsigned 43 is 00101011.
2. Because the original value was negative, we form the two’s complement of 00101011, which is 11010101. This is the representation of \(-43\) decimal.

Converting Signed Decimal to Hexadecimal

To convert a signed decimal integer to hexadecimal, do the following:

1. Convert the absolute value of the decimal integer to hexadecimal.
2. If the decimal integer was negative, form the two’s complement of the hexadecimal number from the previous step.

Converting Signed Hexadecimal to Decimal

To convert a signed hexadecimal integer to decimal, do the following:

1. If the hexadecimal integer is negative, form its two’s complement; otherwise, retain the integer as is.
2. Using the integer from the previous step, convert it to decimal. If the original value was negative, attach a minus sign to the beginning of the decimal integer.

You can tell whether a hexadecimal integer is positive or negative by inspecting its most significant (highest) digit. If the digit is \(\geq 8\), the number is negative; if the digit is \(\leq 7\), the number is positive. For example, hexadecimal 8A20 is negative and 7FD9 is positive.

Maximum and Minimum Values

A signed integer of \(n\) bits uses only \(n - 1\) bits to represent the number’s magnitude. Table 1-7 shows the minimum and maximum values for signed bytes, words, doublewords, and quadwords.

1.3.6 Character Storage

If computers can store only binary data, how do they represent characters? A character set is required, a mapping of characters to integers. Until a few years ago, character sets used only 8 bits. Even now, when running in character mode (such as MS-DOS), IBM-compatible microcomputers use the ASCII (pronounced “askay”) character set. ASCII is an acronym for American Standard Code for Information Interchange. In ASCII, a unique 7-bit integer is assigned to each character. Because ASCII codes use only the lower 7 bits of every byte, the extra bit is used on various computers to create a proprietary character set. On IBM-compatible microcomputers, for example, values 128 through 255 represent graphics symbols and Greek characters.

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Range (Low–High)</th>
<th>Powers of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed byte</td>
<td>(-128) to (+127)</td>
<td>(-2^7) to ((2^7 - 1))</td>
</tr>
<tr>
<td>Signed word</td>
<td>(-32,768) to (+32,767)</td>
<td>(-2^{15}) to ((2^{15} - 1))</td>
</tr>
<tr>
<td>Signed doubleword</td>
<td>(-2,147,483,648) to (+2,147,483,647)</td>
<td>(-2^{31}) to ((2^{31} - 1))</td>
</tr>
<tr>
<td>Signed quadword</td>
<td>(-9,223,372,036,854,775,808) to (+9,223,372,036,854,775,807)</td>
<td>(-2^{63}) to ((2^{63} - 1))</td>
</tr>
</tbody>
</table>
ANSI Character Set  American National Standards Institute (ANSI) defines an 8-bit character set used to represent up to 256 characters. The first 128 characters correspond to the letters and symbols on a standard U.S. keyboard. The second 128 characters represent special characters such as letters in international alphabets, accents, currency symbols, and fractions. MS-Windows Me, 98, and 95 use the ANSI character set. To increase the number of available characters, MS-Windows switches between character tables known as code pages.

Unicode Standard  There has been a need for some time to represent a wide variety of international languages in computer software and to avoid the clutter of hundreds of diverse coding schemes in existence. As a result, the Unicode standard was created as a universal way of defining characters and symbols. It defines codes for characters, symbols, and punctuation used in all major languages, as well as European alphabetic scripts, Middle Eastern right-to-left scripts, and many scripts of Asia. Three encoding forms are available in Unicode, permitting data to be transmitted in byte, word, or doubleword formats:

- UTF-8 is used in HTML, and has the same byte values as ASCII (American Standard Code for Information Interchange). It can be incorporated into a variable-length encoding system for all Unicode characters.
- UTF-16 is used in environments that balance efficient access to characters with economical use of storage. Windows NT, 2000, and XP, for example, use UTF-16 encoding. Each character is encoded in 16 bits.
- UTF-32 is used in environments where space is no concern and fixed-width characters are required. Each character is encoded in 32 bits.

You can copy a smaller Unicode value (byte, for example) into a larger one (word or doubleword) without losing any data.

ASCII Strings  A sequence of one or more characters is called a string. More specifically, an ASCII string is stored in memory as a succession of bytes containing ASCII codes. For example, the numeric codes for the string “ABC123” are 41h, 42h, 43h, 31h, 32h, and 33h. A null-terminated string is a string of characters followed by a single byte containing zero. The C and C++ languages use null-terminated strings, and many MS-DOS and MS-Windows functions require strings to be in this format.

Using the ASCII Table  A table on the inside back cover of this book lists ASCII codes used when running in MS-DOS mode. To find the hexadecimal ASCII code of a character, look along the top row of the table and find the column containing the character you want to translate. The most significant digit of the hexadecimal value is in the second row at the top of the table; the least significant digit is in the second column from the left. For example, to find the ASCII code of the letter a, find the column containing the a and look in the second row: The first hexadecimal digit is 6. Next, look to the left along the row containing a and note that the second column contains the digit 1. Therefore, the ASCII code of a is 61 hexadecimal. This is shown as follows in simplified form:

ASCII Control Characters  Character codes in the range 0 through 31 are called ASCII control characters. If a program writes these codes to standard output (as in C++), the control characters will carry out predefined actions. Table 1-8 lists the most commonly used characters in this range.
Terminology for Numeric Data Representation

It is important to use precise terminology when describing the way numbers and characters are represented in memory and on the display screen. Decimal 65, for example, is stored in memory as a single binary byte as 01000001. A debugging program would probably display the byte as “41,” which is the number’s hexadecimal representation. If the byte were copied to video memory, the letter “A” would appear on the screen. Why? Because 01000001 is the ASCII code for the letter A. Because a number’s interpretation can depend on the context in which it appears, we assign a specific name to each type of data representation to clarify future discussions:

- A binary integer is an integer stored in memory in its raw format, ready to be used in a calculation. Binary integers are stored in multiples of 8 bits (8, 16, 32, 48, or 64).
- An ASCII digit string is a string of ASCII characters, such as “123” or “65,” which is made to look like a number. This is simply a representation of the number and can be in any of the formats shown for the decimal number 65 in Table 1-9:

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII binary</td>
<td>“01000001”</td>
</tr>
<tr>
<td>ASCII decimal</td>
<td>“65”</td>
</tr>
<tr>
<td>ASCII hexadecimal</td>
<td>“41”</td>
</tr>
<tr>
<td>ASCII octal</td>
<td>“101”</td>
</tr>
</tbody>
</table>

### Table 1-8 ASCII Control Characters.

<table>
<thead>
<tr>
<th>ASCII Code (Decimal)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Backspace (moves one column to the left)</td>
</tr>
<tr>
<td>9</td>
<td>Horizontal tab (skips forward n columns)</td>
</tr>
<tr>
<td>10</td>
<td>Line feed (moves to next output line)</td>
</tr>
<tr>
<td>12</td>
<td>Form feed (moves to next printer page)</td>
</tr>
<tr>
<td>13</td>
<td>Carriage return (moves to leftmost output column)</td>
</tr>
<tr>
<td>27</td>
<td>Escape character</td>
</tr>
</tbody>
</table>

### Table 1-9 Types of Numeric Strings.

1. Explain the term LSB.
2. Explain the term MSB.
3. What is the decimal representation of each of the following unsigned binary integers?
   a. 11111000
   b. 11001010
   c. 11110000
4. What is the decimal representation of each of the following unsigned binary integers?
   a. 00110101
   b. 10010110
   c. 11001100
5. What is the sum of each pair of binary integers?
   a. 00001111 + 00000010
b. 11010101 + 01101011
   c. 00001111 + 00001111

6. What is the sum of each pair of binary integers?
   a. 10101111 + 11011011
   b. 10010111 + 11111111
   c. 01110101 + 10101100

7. How many bytes are in each of the following data types?
   a. word
   b. doubleword
   c. quadword

8. How many bits are in each of the following data types?
   a. word
   b. doubleword
   c. quadword

9. What is the minimum number of binary bits needed to represent each of the following unsigned decimal integers?
   a. 65
   b. 256
   c. 32768

10. What is the minimum number of binary bits needed to represent each of the following unsigned decimal integers?
    a. 4095
    b. 65534
    c. 2134657

11. What is the hexadecimal representation of each of the following binary numbers?
    a. 1100 1111 0101 0111
    b. 0101 1100 1010 1101
    c. 1001 0011 1110 1011

12. What is the hexadecimal representation of each of the following binary numbers?
    a. 0011 0101 1101 1010
    b. 1100 1110 1010 0011
    c. 1111 1110 1101 1011

13. What is the binary representation of the following hexadecimal numbers?
    a. E5B6AED7
    b. B697C7A1
    c. 234B6D92

14. What is the binary representation of the following hexadecimal numbers?
    a. 0126F9D4
    b. 6ACDFA95
    c. F69BDC2A

15. What is the unsigned decimal representation of each hexadecimal integer?
    a. 3A
    b. 1BF
    c. 4096

16. What is the unsigned decimal representation of each hexadecimal integer?
    a. 62
b. 1C9
c. 6A5B

17. What is the 16-bit hexadecimal representation of each signed decimal integer?
   a. −26
   b. −452

18. What is the 16-bit hexadecimal representation of each signed decimal integer?
   a. −32
   b. −62

19. The following 16-bit hexadecimal numbers represent signed integers. Convert to decimal.
   a. 7CAB
   b. C123

20. The following 16-bit hexadecimal numbers represent signed integers. Convert to decimal.
   a. 7F9B
   b. 8230

21. What is the decimal representation of the following signed binary numbers?
   a. 10110101
   b. 00101010
   c. 11110000

22. What is the decimal representation of the following signed binary numbers?
   a. 10000000
   b. 11001100
   c. 10110111

23. What is the 8-bit binary (two’s-complement) representation of each of the following signed decimal integers?
   a. −5
   b. −36
   c. −16

24. What is the 8-bit binary (two’s-complement) representation of each of the following signed decimal integers?
   a. −72
   b. −98
   c. −26

25. What are the hexadecimal and decimal representations of the ASCII character capital X?

26. What are the hexadecimal and decimal representations of the ASCII character capital M?

27. Why was Unicode invented?

28. Challenge: What is the largest value you can represent using a 256-bit unsigned integer?

29. Challenge: What is the largest positive value you can represent using a 256-bit signed integer?

1.4 Boolean Operations

Boolean algebra defines a set of operations on the values true and false. It was invented by George Boole, a mid–nineteenth-century mathematician who designed the first model of a computer (named the Analytical Engine). When early digital computers were invented, it was clear that Boole’s algebra could be used to describe the design of digital circuits. At the same time, boolean expressions are used in programming to express logical operations.

Boolean Expression A boolean expression involves a boolean operator and one or more operands. Each boolean expression implies a value of true or false. The set of operators includes the following:

- NOT: notated as ¬ or ~ or ’
• AND: notated as $\land$ or $\cdot$
• OR: notated as $\lor$ or $+$

The NOT operator is unary, and the other operators are binary. The operands of a boolean expression can also be boolean expressions. The following are examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg X$</td>
<td>NOT $X$</td>
</tr>
<tr>
<td>$X \land Y$</td>
<td>$X$ AND $Y$</td>
</tr>
<tr>
<td>$X \lor Y$</td>
<td>$X$ OR $Y$</td>
</tr>
<tr>
<td>$\neg (X \land Y)$</td>
<td>(NOT $X$) OR $Y$</td>
</tr>
<tr>
<td>$\neg (X \lor Y)$</td>
<td>NOT ($X$ AND $Y$)</td>
</tr>
<tr>
<td>$X \land \neg Y$</td>
<td>$X$ AND (NOT $Y$)</td>
</tr>
</tbody>
</table>

**NOT** The NOT operation reverses a boolean value. It can be written in mathematical notation as $\neg X$, where $X$ is a variable (or expression) holding a value of true (T) or false (F). The following truth table shows all the possible outcomes of NOT using a variable $X$. Inputs are on the left side and outputs (shaded) are on the right side:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\neg X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

A truth table can use 0 for false and 1 for true.

**AND** The Boolean AND operation requires two operands, and can be expressed using the notation $X \land Y$. The following truth table shows all the possible outcomes (shaded) for the values of $X$ and $Y$:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \land Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

The output is true only when both inputs are true. This corresponds to the logical AND used in compound boolean expressions in C++ and Java.

**OR** The Boolean OR operation requires two operands, and is often expressed using the notation $X \lor Y$. The following truth table shows all the possible outcomes (shaded) for the values of $X$ and $Y$:

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>$X \lor Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
The output is false only when both inputs are false. This truth table corresponds to the logical OR used in compound boolean expressions in C++ and Java.

**Operator Precedence** In a boolean expression involving more than one operator, precedence is important. As shown in the following table, the NOT operator has the highest precedence, followed by AND and OR. To avoid ambiguity, use parentheses to force the initial evaluation of an expression:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Order of Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬X ∨ Y</td>
<td>NOT, then OR</td>
</tr>
<tr>
<td>¬(X ∨ Y)</td>
<td>OR, then NOT</td>
</tr>
<tr>
<td>X ∨ (Y ∧ Z)</td>
<td>AND, then OR</td>
</tr>
</tbody>
</table>

### 1.4.1 Truth Tables for Boolean Functions

A boolean function receives boolean inputs and produces a boolean output. A truth table can be constructed for any boolean function, showing all possible inputs and outputs. The following are truth tables representing boolean functions having two inputs named X and Y. The shaded column on the right is the function’s output:

**Example 1:** ¬X ∨ Y

<table>
<thead>
<tr>
<th>X</th>
<th>¬X</th>
<th>Y</th>
<th>¬X ∨ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**Example 2:** X ∧ ¬Y

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>¬Y</th>
<th>X ∧ ¬Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Example 3: \((Y \land S) \lor (X \land \neg S)\)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>S</th>
<th>Y \land S</th>
<th>\neg S</th>
<th>X \land \neg S</th>
<th>(Y \land S) \lor (X \land \neg S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

This boolean function describes a multiplexer, a digital component that uses a selector bit \(S\) to select one of two outputs \((X\) or \(Y)\). If \(S = false\), the function output \((Z)\) is the same as \(X\). If \(S = true\), the function output is the same as \(Y\). Here is a block diagram of a multiplexer:

1.4.2 Section Review

1. Describe the following boolean expression: \(\neg X \lor Y\).
2. Describe the following boolean expression: \((X \land Y)\).
3. What is the value of the boolean expression \((T \land F) \lor T\) ?
4. What is the value of the boolean expression \(\neg(F \lor T)\) ?
5. What is the value of the boolean expression \(\neg F \lor \neg T\) ?
6. Create a truth table to show all possible inputs and outputs for the boolean function described by \(\neg(A \lor B)\).
7. Create a truth table to show all possible inputs and outputs for the boolean function described by \((\neg A \land \neg B)\).
8. Challenge: If a boolean function has four inputs, how many rows would be required for its truth table?
9. Challenge: How many selector bits would be required for a four-input multiplexer?

1.5 Chapter Summary

This book focuses on programming microprocessors compatible with the Intel IA-32 processor family, using the MS-Windows platform.

We cover basic principles about computer architecture, machine language, and low-level programming. You will learn enough assembly language to test your knowledge on today's most widely used microprocessor family.
Before reading this book, you should have completed a single college course or equivalent in computer programming.

An assembler is a program that converts source-code programs from assembly language into machine language. A companion program, called a linker, combines individual files created by an assembler into a single executable program. A third program, called a debugger, provides a way for a programmer to trace the execution of a program and examine the contents of memory.

You will create two basic types of programs: 16-bit real-address mode programs and 32-bit protected mode programs.

You will learn the following concepts from this book: basic computer architecture applied to Intel IA-32 processors; elementary boolean logic; how IA-32 processors manage memory; how high-level language compilers translate statements from their language into assembly language and native machine code; how high-level languages implement arithmetic expressions, loops, and logical structures at the machine level; and the data representation of signed and unsigned integers, real numbers, and character data.

Assembly language has a one-to-one relationship with machine language, in which a single assembly language instruction corresponds to one machine language instruction. Assembly language is not portable because it is tied to a specific processor family.

Languages are tools you apply to individual applications or parts of applications. Some applications, such as device drivers and hardware interface routines, are more suited to assembly language. Other applications, such as multiplatform business applications, are suited to high-level languages.

The virtual machine concept is an effective way of showing how each layer in a computer architecture represents an abstraction of a machine. Layers can be constructed of hardware or software, and programs written at any layer can be translated or interpreted by the next-lowest layer. The virtual machine concept can be related to real-world computer layers, including digital logic, microarchitecture, instruction set architecture, operating system, assembly language, and high-level languages.

Binary and hexadecimal numbers are essential notational tools for programmers working at the machine level. For this reason, you must understand how to manipulate and translate between number systems and how character representations are created by computers.

The following boolean operators were presented in this chapter: NOT, AND, and OR. A boolean expression combines a boolean operator with one or more operands. A truth table is an effective way to show all possible inputs and outputs of a boolean function.

End Notes

3. Its source code might have been written in C or assembly language, but once compiled, the operating system is simply a Level 2 program that interprets Level 3 commands.
6. You can see a working model of Boole’s Analytical Engine in the London Science Museum.