1

Basic Concepts

1.1 Welcome to Assembly Language

This book, entitled *Assembly Language for Intel-Based Computers*, focuses on programming Intel microprocessors, specifically members of the Intel IA-32 processor family. The IA-32 family began with the Intel 80386, and continues on through the current Pentium 4. Assembly language is the oldest programming language, and of all languages, it bears the closest resemblance to the native language of a computer. It provides direct access to a computer's hardware, making it necessary for you to understand a great deal about your computer's architecture and operating system.

Educational Value Why do you have to 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

In fact, these are names of actual courses at colleges and universities that used the third edition of this book. You will probably find that this book contains more assembly language techniques, reference information, and examples than you can possibly digest in a single semester.

If you are in a course whose name includes either the word *architecture* or *fundamentals*, this book will give you some basic principles about computer architecture, machine language, and low-level programming that will stay with you for years to come. 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; this is the real thing, the same one used by the professionals. You will learn the architecture of the Intel IA-32 processor family from the programmer's point of view.

If you are in doubt about the value of spending endless hours studying the low-level details of computer software and hardware, perhaps you can find inspiration in the following quote from a lecture given by one of the greatest computer scientists of our time, Donald Knuth:

> "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]

Web Site Before you go any farther, visit the book's Web site to see the extra support information and workbook exercises you can use:

```
http://www.nuvisionmiami.com/books/asm
```

There are always new workbook tutorials, interesting example programs, corrections to errors in the text, and so on. If for some reason the given URL is not available, you can reach the book's Web site through Prentice Hall's URL (www.prenhall.com). Search for "Kip Irvine".

1.1.1 Some Good Questions to Ask

Maybe we can answer some of your questions about this book and how it can be used.

What background should I have? Before reading this book, you should have completed a single college course or its equivalent in computer programming. Most students learn C++, C#, Java, or Visual Basic. Other languages will work, provided they have similar features.

1. Donald Knuth: *MMIX, A RISC Computer for the New Millennium*, Transcript of a lecture given at the Massachussetts Institute of Technology, December 30, 1999.

What is an assembler? An *assembler* is a program that converts source-code programs from assembly language into machine language. The assembler can optionally generate a source listing file with line numbers, memory addresses, source code statements, and a cross-reference listing of symbols and variables used in a program. 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. Two of the most popular assemblers for the Intel family are MASM (Microsoft Assembler) and TASM (Borland Turbo Assembler).

What hardware and software do I need? You need a computer with an Intel386, Intel486, or one of the Pentium processors. All of these belong to the IA-32 processor family, as Intel calls it. Your operating system may be some version of Microsoft Windows, MS-DOS, or even Linux running a DOS emulator. The following are either required or recommended:

- ***Editor:*** You need a simple text editor that can create assembly language source files. You can use TextPad by Helios Software, which is supplied on the CD-ROM with this book. Or you can use NotePad (free with Windows), or the Microsoft Visual Studio editor (used with Visual C++). Any other editor that produces plain ASCII text files will do also.
- ***Assembler:*** You need Microsoft Assembler (MASM) Version 6.15, supplied free with this book on a CD-ROM. Update patches, as they become available, can be downloaded from the Microsoft Web site. If you absolutely must use the Borland Turbo Assembler, you'll be happy to know that Borland-compatible versions of the book's example programs are available on the author's Web site.
- ***Linker:*** You need a linker utility to produce executable files. We supply two linkers on the CD-ROM with this book: The Microsoft 16-bit linker, named LINK.EXE, and the Microsoft 32-bit linker, named LINK32.EXE.
- ***Debugger:*** Strictly speaking, you don't need a debugger, but you will probably want one. For MS-DOS programs, MASM supplies a good 16-bit debugger named *CodeView.* TASM supplies one named *Turbo Debugger.* For 32-bit Windows Console programs, our preferred debugger is Microsoft Visual Studio (msdev.exe), part of Microsoft Visual C++.

What types of programs will I create? You will create two basic types of programs:

- ***16-Bit Real-Address Mode:*** If you're running either pure MS-DOS or a DOS emulator, you can create 16-bit Real-address mode programs. Most of the programs in this book have been adapted for your use, in special directories named **RealMode** on the disk accompanying this book. 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 under MS-DOS.
- ***32-Bit Protected Mode:*** If you're using Microsoft Windows, you can create 32-bit Protected mode programs that display both text and graphics.

What do I get with this book? You get a lot of printed paper. On the CD attached to the book, you get a complete copy of the Microsoft Assembler, version 6.15. You get a collection of example programs on the CD. Best of all, you get a whole lot of information on the author's Web site, including:

- Updates to the example programs. No doubt some of the programs will be improved and corrected.
- The *Assembly Language Workbook*, a constantly expanding collection of practice exercises covering topics from all over the book.
- Complete source code for the book's link libraries. One library is for 32-bit Protected mode under MS-Windows; the other library is for Real-address mode programming under MS-DOS or a DOS emulator. (*Note:* MS-Windows can also run Real-address mode programs.)
- Corrections to the book. Hopefully there won't be too many of these!
- Helpful hints on installing the assembler and configuring different editors to run it. Two editors I currently use are Microsoft Visual C++ and *TextPad* by Helios Software.
- Frequently asked questions. In the previous edition, there were about 40 of these.
- Additional topics on MS-Windows programming, graphics programming, and so on, that could not be included in the printed book for lack of space.
- E-mail access to the author for corrections and clarifications directly related to the book. But don't ask me to help you debug your programming projects. That's your professor's job.
- Solutions to programming exercises. In the previous editions, only professors were given access to solution programs, but this turned out to be somewhat controversial. I was continually fending off e-mail requests for solutions by individuals who (said they) were self-studying assembly language. (There will be additional suggested programming assignments posted on the instructor Web, which will *absolutely, positively* be available only to registered college instructors.)

What will I learn? Here are some of the ways this book will make you better informed about computer architecture, programming, and computer science:

- You will learn some basic principles of computer architecture, as applied to the Intel IA-32 processor family.
- You will learn some basic boolean logic and how it applies to programming and computer hardware.
- You will learn about how IA-32 processors manage memory, using real mode, protected mode, and virtual mode.
- You will learn how high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code.
- You will learn how high-level languages implement arithmetic expressions, loops, and logical structures at the machine level.

- You will learn about data representation, including signed and unsigned integers, real numbers, and character data.
- You will improve your machine-level debugging skills. Even in C++, when your programs have errors due to pointers or memory allocation, you can dive to the machine level and find out what really went wrong. High-level languages purposely hide machine-specific details, but sometimes these details are important when tracking down errors.
- You will learn how application programs communicate with the computer's operating system via interrupt handlers, system calls, and common memory areas. You will also learn how the operating system loads and executes application programs.
- You will learn how to interface assembly language code to C++ programs.
- You will gain the confidence to write new assembly language programs without having to ask anyone for help.

How does assembly language relate to machine language? First, *machine language* is a numeric language that is specifically understood by a computer's processor (the CPU). Intel processors, for example, have a machine language that is automatically understood by other Intel processors. Machine language consists purely of numbers.

Assembly language consists of statements that use short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* relationship with machine language, meaning that *one* assembly language instruction corresponds to *one* 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 both assembly language and machine language. A single statement in C++, for example, expands into multiple assembly language or machine instructions.

Let's find out first-hand how C++ statements expand into machine code. Most people cannot read raw machine code, so we will show its closest relative, assembly language, instead. The following C++ statement carries out two arithmetic operations and assigns the result to a variable. Assume that X and Y are integers:

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

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

```
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 inside the CPU which are often used for intermediate results of operations.)

The point in this example is not to show that C++ is "better" or more powerful than assembly language, but to show how assembly language implements a statement in a high-level language. The assembly language statements have a one-to-one correspondence with the computer's native machine language, which is a set of coded numbers with special meaning to the processor.

> ***We? Who's that?*** Throughout this book, you're going to see constant references to *we*. Authors of textbooks and academic articles often use *we* as a formal reference to themselves. It just seems too informal to say, "I will now show you how to" do such-and-such. If it helps, think of *we* as a reference to the author, his reviewers (who really helped him a lot), his publisher (Prentice-Hall), and his students (thousands).

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 are 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 that only exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Assembly language, on the other hand, makes no attempt to be portable. It is tied to a specific processor family, so there are a number of different assembly languages widely used today. Each is based on either a processor family or a specific computer, with names such as Motorola 68x00, Intel IA-32, SUN Sparc, Vax, and IBM-370. The instructions in assembly language match the computer's instruction set architecture. For example, the assembly language taught in this book works only on processors belonging to the Intel IA-32 family.

Why learn assembly language? Why not just read a good book on computer hardware and architecture, and avoid having to learn assembly language programming?

- You may be working toward a degree in computer engineering. If so, there is a strong likelihood that you will write *embedded systems* programs. Such programs are written in C, Java, or assembly language, and downloaded into computer chips and installed in dedicated devices. Some examples are automobile fuel and ignition systems, air-conditioning control systems, security systems, flight control systems, hand-held computers, modems, printers, and other intelligent computer peripherals.
- Many dedicated computer game machines have stringent memory restrictions, requiring programs to be highly optimized for both space and runtime speed. Game programmers are experts at writing code that takes full advantage of specific hardware features in a target system. They frequently use assembly language as their tool of choice because it permits total control over the creation of machine code.
- If you are working toward a degree in computer science, assembly language will help you gain an overall understanding of the interaction between the computer hardware, operating

system, and application programs. Using assembly language, you can apply and test the theoretical information you are given in computer architecture and operating systems courses.

- If you're working as an application programmer, you may find that limitations in your current language prevent you from performing certain types of operations. For example, Microsoft Visual Basic doesn't handle character processing very efficiently. Programmers generally rely on DLL (*dynamic link libraries)* written in C++ or assembly language to perform character operations such as data encryption and bit manipulation.
- If you work for a hardware manufacturer, you may have to create device drivers for the equipment you 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 any rules in assembly language? Yes, there are a few rules, mainly due to the physical limitations of the processor and its native instruction set. Two operands used in the same instruction, for example, must be the same size. But assembly language is far less restricting than C++.

Assembly language programs can easily bypass restrictions characteristic of high-level languages. For example, the C++ language does not allow a pointer of one type to be assigned to a pointer of another type. Ordinarily, this is a good restriction because it helps avoid logic errors in programs. An experienced programmer can find a way around this restriction but in doing so may end up writing code that is overly tricky. Assembly language, in contrast, has no restriction regarding pointers. The assignment of a pointer is left to the programmer's discretion. Of course, the price for such freedom is high: an assembly language programmer spends a lot of time debugging programs at the machine level.

1.1.2 Assembly Language Applications

In the early days of programming, most application programs were written partially or entirely in assembly language because programs had to fit in a small area of memory and had to run as efficiently as possible. As computers became more powerful, programs became longer and more complex; this demanded the use of high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability to assist the programmer. More recently, object-oriented languages such as C++, C#, Visual Basic, and Java have made it possible to write complex programs containing millions of lines of code.

It is rare to see large application programs written 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. Assembly language is also used when writing embedded systems programs and device

drivers. Table 1–1 compares the adaptability of assembly language to high-level languages in relation to various types of computer programs.

Table 1–1 Comparison of Assembly Language to High-Level Languages.

Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

C++ has the unique quality of offering a compromise between high-level structure and low-level details. Direct hardware access is possible but completely non-portable. 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 the assembler and linker 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. Is type checking on pointer variables stronger 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 that directly accesses 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 * 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*.[2] To explain this concept, let us begin with the most basic function of a computer, that of executing programs.

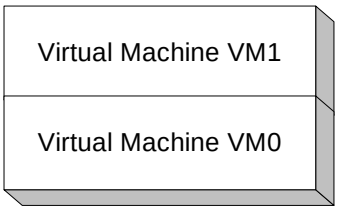
A computer is ordinarily constructed in such a way that it directly executes programs written in what may be called its *machine language*. Each instruction in this language is simple enough that it can be executed using a relatively small number of electronic circuits. For simplicity, we will call this language **L0**.

But 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 thinking purely in terms of languages, Tanenbaum suggests thinking 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, as shown below:

2. Andrew S. Tanenbaum. *Structured Computer Organization*, 4th Edition. 1999, Prentice-Hall.



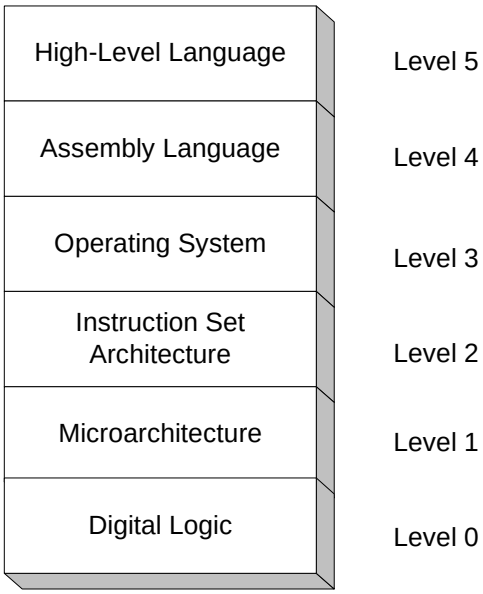
Each virtual machine can be constructed of either hardware or software. People can write programs for virtual machine VM1, and if it is practical to implement VM1 as an actual computer, programs can be executed directly on the hardware. Or, programs written in VM1 can be interpreted/translated and executed 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 which is more easily understood. This process can repeat itself until a virtual machine VM*n* can be designed that supports 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 that is 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 like **Level 1** for VM1, and **Level 0** for VM0, shown in Figure 1–1. Let us assume that a computer's digital logic hardware represents machine Level 0, and that Level 1 is implemented by an interpreter hard-wired 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.

Figure 1–1 Virtual Machine Levels 0 through 5.



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 microinstructions to carry out a primitive operation such as fetching a number from memory and incrementing it by 1.

Instruction Set Architecture (Level 2) Computer chip manufacturers design into the processor an *instruction set* that can be used 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.[3]

Assembly Language (Level 4) Above the operating system level, programming languages provide the translation layers that make large-scale software development practical. Assembly

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.

language, which appears at Level 4, uses short mnemonics such as ADD, SUB, and MOV that 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 usually translated (*assembled*) in their entirety into machine language before they begin to execute.

High-Level Languages (Level 5) At Level 5 are languages such as C++, C#, Visual Basic, and Java. Programs in these languages contain powerful statements that often translate into multiple instructions at Level 4. Most C++ debuggers, for example, have the option to view a window that lists the assembly language translation of your code. In Java, you would look at a symbolic listing of *Java byte code* to see the same sort of translation. Level 5 programs are usually translated by compilers into Level 4 programs. They usually have a built-in assembler that immediately translates the Level 4 code into conventional machine language.

> The Intel IA-32 processor architecture supports multiple virtual machines. Its *virtual-86* operating mode emulates the architecture of the Intel 8086/8088 processor, used in the original IBM Personal Computer. The Pentium can run multiple instances of the virtual-86 machine at the same time, so that independent programs running on each virtual machine seem to have complete control of their host computer.

1.2.1 The History of PC Assemblers

There is no universal assembly language specification for Intel processors. What has emerged over the years is a *de facto* standard, established by Microsoft's popular MASM Version 5 assembler. Borland International established itself as a major competitor in the early 1990s with TASM (Turbo Assembler). TASM added many enhancements, producing what was called *Ideal Mode*, and Borland also provided a *MASM compatibility mode* which matched the syntax of MASM Version 5.

Microsoft released MASM 6.0 in 1992, which was a major upgrade with many new features. Since that time, Microsoft has released minor upgrades in versions 6.11, 6.13, 6.14, and 6.15, to keep up with changes to each new Pentium instruction set. The assembler syntax has not changed since version 6.0. Borland released 32-bit TASM 5.0 in 1996, which matches the MASM 6.0 syntax.

There are other popular assemblers, all of which vary from MASM's syntax to a greater or lesser degree: To name a few, there are: NASM (Netwide Assembler) for both Windows and Linux, MASM32, a shell built on top of MASM, Asm86, and GNU assembler, distributed by the Free Software Foundation.

1.2.2 Section Review

1. In your own words, describe the *virtual machine* concept.
2. Why don't programmers use the native language of a computer to write application programs?
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 demonstrate 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 we can begin to discuss computer organization and assembly language, we need a common mode of communication with numbers. Specifically, computer data can be represented in a variety of ways. Because we are dealing with the computer at the machine level, it is necessary to examine the contents of memory and registers. Computers are constructed from digital circuits that have only two states: *on* and *off*. At times, we will use binary numbers to describe the contents of computer memory; at other times, decimal and hexadecimal numbers will be used. You must develop a certain fluency with number formats, and have the ability to 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 then continue with the letters A through F to represent decimal values 10

through 15. It's quite common to use hexadecimal numbers when showing the contents of computer memory and machine-level instructions.

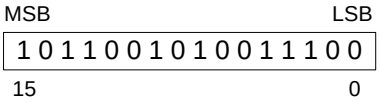
Table 1–2 Binary, Octal, Decimal, and Hexadecimal Digits

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

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:



Binary integers can be either signed or unsigned. A signed integer can be either positive or negative. An unsigned integer can only be positive, including zero. Through special encoding schemes, binary numbers can even represent real numbers. For now, we begin with the simplest type of binary numbers, unsigned integers.

1.3.1.1 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 the powers of two increase from right to left:

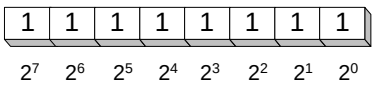


Table 1–3 lists the decimal values of 2^0 through 2^{15}.

Table 1–3 Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

1.3.1.2 Translating Unsigned Binary Integers to Decimal

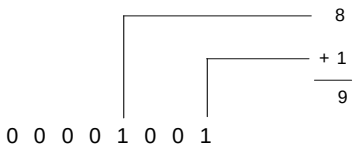
Weighted positional notation represents a convenient way to calculate the decimal value of an unsigned binary integer having *n* digits:

$$dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + ... + (D_1 \times 2^1) + (D_0 \times 2^0)$$

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

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

This is also shown in the following figure:



(F009)

1.3.1.3 Translating Unsigned Decimal Integers to Binary

To translate an unsigned decimal integer into binary, repeatedly divide the decimal value by 2, saving each remainder as a binary digit. Here is an example of how we would translate decimal 37. The remainder digits, starting from the top row, are the binary digits D_0, D_1, D_2, D_3, D_4, and

D_5:

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

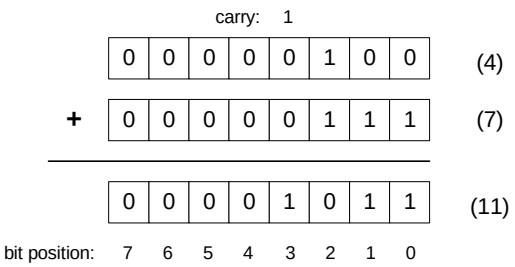
Collecting the binary digits in the remainder column in reverse order produces binary 100101. Because we are used to working with binary numbers whose lengths are multiples of 8, we can fill the remaining two digit positions on the left with zeros, producing 00100101.

1.3.2 Binary Addition

When adding two binary integers, you must proceed bit by bit, beginning with the lowest order pair of bits (on the right side). Each bit pair is added. There are only four ways to add two binary digits, as shown below:

0 + 0 = 0	0 + 1 = 1
1 + 0 = 1	1 + 1= 10

In one case, when adding 1 to 1, the result is 10 binary. (You can also think of this as the decimal value 2.) The extra digit generates a carry to the next-highest bit position. In the following figure, for example, we add binary 00000100 to 00000111:



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:

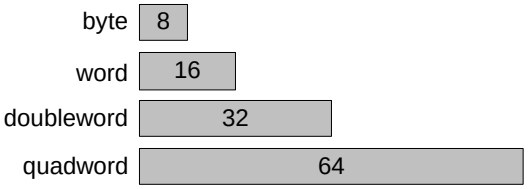


Table 1–4 shows the range possible values for each type of unsigned integer.

Table 1–4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Large Measurements A number of large measurements are used when referring to both memory and disk space:[4]

- One *kilobyte* is equal to 2^{10}, or 1,024 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.
- 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}.
- One *yottabyte* is equal to 2^{80}.

4. Source: www.webopedia.com.

1.3.4 Hexadecimal Integers

Large binary numbers are cumbersome to read, so hexadecimal digits are usually used by assemblers and debuggers 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 can have a value from 0 to 15, so the letters A to F are used, as well as the digits 0-9. The letter A = 10, B = 11, C = 12, D = 13, E = 14, and F = 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.

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

In the following example, we can see that the binary integer 000101101010011110010100 is represented by hexadecimal 16A794:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

It is often useful to display binary integers with a space between each group of four bits. Translation from binary to hexadecimal becomes that much easier.

1.3.4.1 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. First, let's number the digits in a 4-digit hexadecimal integer with subscripts as $D_3D_2D_1D_0$. The following formula calculates the number's decimal value:

$$dec = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

This can be generalized for any *n*-digit hexadecimal number:

$$dec = (D_{n-1} \times 16^{n-1}) + (D_{n-2} \times 16^{n-2}) + \ldots + (D_1 \times 16^1) + (D_0 \times 16^0)$$

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 4,660. Similarly, hexadecimal 3BA4 is equal to $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268. The following figure shows this last calculation:

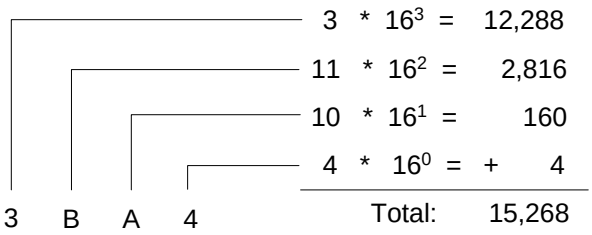


Table 1–6 lists the powers of 16, from 16^0 to 16^7.

Table 1–6 Powers of 16, in Decimal.

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

1.3.4.2 Converting Unsigned Decimal to Hexadecimal

To convert an unsigned decimal integer to hexadecimal, repeatedly divide the decimal value by 16, and keep each remainder as a hexadecimal digit. For example, in the following table, we convert decimal 422 to hexadecimal:

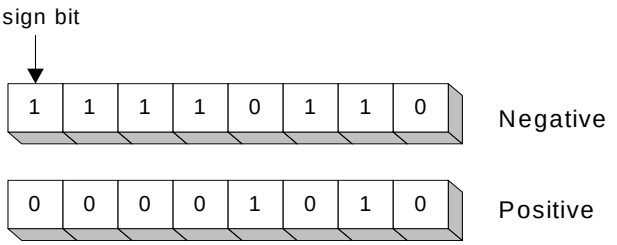
Division	Quotient	Remainder
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

If we collect the digits from the remainder column in reverse order, the hexadecimal representation is **1A6**. You may recall that we used the same algorithm for binary numbers back in

Section 1.3.1.2. It works for any number base, just by changing the divisor.

1.3.5 Signed Integers

As we said earlier, signed binary integers can be either positive or negative. In general, the most significant bit (MSB) indicates the number's sign. A value of 0 indicates that the integer is positive, and 1 indicates that it is negative. For example, the following figure shows examples of both negative and positive integers stored in a single byte:



1.3.5.1 Two's Complement Notation

Negative integers are represented using what is called *two's complement* representation. The two's complement of an integer is simply its additive inverse. (You may recall that when a number's *additive inverse* is added to the number, their 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 reversing 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 below.

Starting value	`00000001`
Step 1: reverse the bits	`11111110`
Step 2: add 1 to the value from Step 1	`11111110` `+00000001`
Sum: two's complement representation	`11111111`

Therefore, 11111111 is the two's complement representation of –1. The two's complement operation is reversible, so if you form the two's complement of 11111111, the result 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 --> 95C2 + 1 --> 95C3
95C3 --> 6A3C + 1 --> 6A3D
21F0 --> DE0F + 1 --> DE10
DE10 --> 21EF + 1 --> 21F0
```

Converting Signed Binary to Decimal Suppose you would like to determine the decimal value of a signed binary integer. Here are the steps to follow:

- If the highest bit is a 1, it is currently stored in two's complement notation. You must form its two's complement a second time to get its positive equivalent. Then you can 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:

Starting value	11110000
Step 1: reverse the bits	00001111
Step 2: add 1 to the value from Step 1	00001111 + 1
Step 3: form the two's complement	00010000
Step 4: convert to decimal	16

Remembering that the original integer (11110000) was negative, we infer that its decimal value was **–16**.

Converting Signed Decimal to Binary Suppose you would like to determine the binary representation of a signed decimal integer. Here are the steps to follow:

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

For example, –43 decimal can be translated to binary as follows:

- The binary representation of unsigned 43 is 00101011.

- 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:

- Convert the absolute value of the decimal integer to hexadecimal.
- 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:

- If the hexadecimal integer is negative, form its two's complement; otherwise, retain the integer as is.
- 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 if a hexadecimal integer is positive or negative by inspecting its most significant (highest) digit. If the digit is >= 8, the number is negative; if the digit is <= 7, the number is positive. For example, hexadecimal 8A20 is negative, and 7FD9 is positive.

1.3.5.2 Maximum and Minimum Values

A signed integer of *n* bits can only use *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.

Table 1–7 Storage Sizes and Ranges of Signed Integers.

Storage Type	Range (low–high)	Powers of 2
Signed byte	–128 to +127	-2^7 to $(2^7 - 1)$
Signed word	–32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	–2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

1.3.6 Character Storage

Assuming that a computer can only store binary data, one might wonder how it could also store characters. To do this, it must support a certain *character set*, which is a mapping of characters to integers. Until a few years ago, character sets used only 8 bits. Because of the great diversity

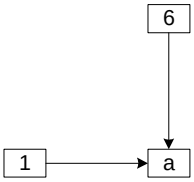
of languages around the world, the 16-bit *Unicode* character set was created to support thousands of different character symbols. [5]

When running in character mode (such as MS-DOS), IBM-compatible microcomputers use the *ASCII* (pronounced “askey”) 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 128–255 represent graphics symbols and Greek characters.

ASCII Strings A sequence of one or more characters is called a *string*. 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 of the MS-DOS and MS-Windows functions require strings to be in this format.

Using the ASCII Table There is a convenient table on the inside back cover of this book that lists all of the ASCII codes 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 that 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 below in simplified form:



MS-Windows programs use a variety of different character sets, so it is not possible to use just a single lookup table. (You can read the Microsoft documentation on Windows fonts to see how characters translate into numeric codes.)

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. Let’s use decimal 65 as an example: stored in memory as a single byte, its binary bit pat-

5. You can read about the Unicode Standard at http:// www.unicode.org.

tern is 01000001. A debugging program would probably display the byte as “41,” which is the hexadecimal notation for this bit pattern. But if the byte were moved to the video display area of memory by a running program, the letter **A** would appear onscreen. This is because 01000001 is the ASCII code for the letter **A**. In other words, the interpretation of numbers on a computer depends greatly on the context in which the number appears.

In this book, we use a naming method for numeric data representation that is reasonably general to avoid conflicts with terminology you might encounter from other sources.

- A *binary number* is a number 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 1–8:

Table 1–8 Types of Numeric Strings.

Format	Value
ASCII binary	"01000001"
ASCII decimal	"65"
ASCII hexadecimal	"41"
ASCII octal	"101"

1.3.7 Section Review

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

In this section we introduce a few fundamental operations of *boolean algebra*, the algebra that defines a set of operations on the values **true** and **false**. This algebra was invented by George Boole, a mid-nineteenth-century mathematician who never saw a working computer. When early computers were designed, it was discovered that his 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:

- NOT: notated as ¬ or ~ or ‘
- AND: notated as ∧ or •
- OR: notated as ∨ 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:

Expression	Description
¬X	NOT X
X ∧ Y	X AND Y
X ∨ Y	X OR Y
¬X ∨ Y	(NOT X) OR Y

Expression	Description
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

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:

X	**$\neg$X**
F	T
T	F

A truth table can just as easily be constructed using 0 for false and 1 for true.

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

X	**Y**	**X $\wedge$ Y**
F	F	F
F	T	F
T	F	F
T	T	T

Note that the output is true only when both inputs are true. This corresponds to the logical AND used in compound boolean expressions in programming languages such as C++ and Java.

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

X	**Y**	**X $\vee$ Y**
F	F	F
F	T	T

X	Y	X ∨ Y
T	F	T
T	T	T

Note that the output is false only when both inputs are false. This corresponds to the logical OR used in compound boolean expressions in programming languages such as C++ and Java.

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

Expression	Order of Operations
¬X ∨ Y	NOT, then OR
¬(X ∨ Y)	OR, then NOT
X ∨ (Y ∧ Z)	AND, then OR

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 that shows 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 side is the function's output:

Example 1: ¬X ∨ Y

X	¬X	Y	¬X ∨ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

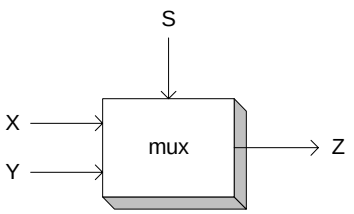
Example 2: X ∧ ¬Y

X	Y	¬Y	X ∧¬Y
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F

Example 3: (Y ∧ S) ∨ (X ∧ ¬S)

X	Y	S	Y ∧ S	¬S	X ∧¬S	(Y ∧ S) ∨ (X ∧ ¬S)
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T

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 outputs is the same as Y. Here is a diagram of such a device:



1.4.2 Section Review

1. Describe the following boolean expression: $\neg X \vee Y$.
2. Describe the following boolean expression: $(X \wedge Y)$.
3. What is the value of the boolean expression $(T \wedge F) \vee T$?
4. What is the value of the boolean expression $\neg(F \vee T)$?
5. What is the value of the boolean expression $\neg F \vee \neg$?
6. Create a truth table to show all possible inputs and outputs for the boolean function described by $\neg(A \vee B)$.
7. Create a truth table to show all possible inputs and outputs for the boolean function described by $(\neg A \wedge \neg B)$.
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, entitled *Assembly Language for Intel-Based Computers*, focuses on programming Intel microprocessors, specifically members of the Intel IA-32 processor family.

This book will give you some 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 its 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; the data representation of signed and unsigned integers, real numbers, and character data.

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

It is important to understand how languages are simply tools that can be applied to various types of applications. Some applications, such as device drivers and hardware interface routines, are more suited to assembly language. Other applications, such as multi-platform business applications, are suited to high-level languages.

The virtual machin*e* 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, it is vital that you understand how to mainpulate and translate between each of the number systems. It is also important to understand 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.