Lecture No.2

Lecture Outlines

1.3 Data Representation

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. You must develop a certain fluency with number formats, so you 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 hardware and software manuals. 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 Integers

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 0 or 1. ***Bits*** are numbered sequentially 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 0 1 1 1 0 0 |
15                                 0   Bit number
```

Table 1-2 Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	1 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

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. When writing down large binary numbers, many people like to insert a dot every 4 bits or 8 bits to make the numbers easier to read. Examples are 1101.1110.0011.1000.0000 and 11001010.10101100.

Unsigned Binary Integers

Starting with the LSB, 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:

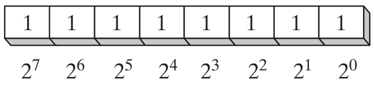


Table 1-3 lists the decimal values of 2^0 through 2^{15}.

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

Translating Unsigned Binary Integers to Decimal

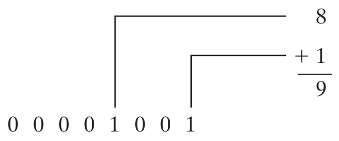
Weighted positional notation represents a convenient way to calculate the decimal value of an unsigned binary integer having *n* digits:

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

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:



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_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

We can concatenate the binary bits from the remainder column of the table in reverse order (D_5, D_4, . . .) to produce binary 100101. Because 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.

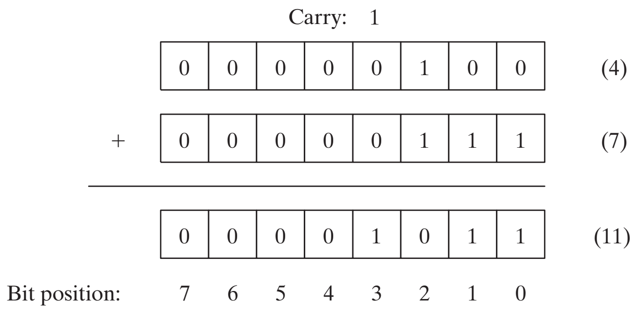
Tip: How many bits? There's a simple formula to find b, the number of binary bits you need to represent the unsigned decimal value n. It is b = ceiling ($\log_2 n$). If n = 17, for example, $\log_2 17$ = 4.087463, which when raised to the smallest following integer, equals 5. Most calculators don't have a log base 2 operation, but you can find web pages that will calculate it for you.

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:

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

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:



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).

Sometimes a carry is generated out of the highest bit position. When that happens, the size of the storage area set aside becomes important. If we add 11111111 to 00000001, for example, a 1 carries out of the highest bit position, and the lowest 8 bits of the sum equal all zeros. If the storage location for the sum is at least 9 bits long, we can represent the sum as 100000000. But if the sum can only store 8 bits, it will equal to 00000000, the lowest 8 bits of the calculated value.

1.3.3 Integer Storage Sizes

The basic storage unit for all data in an x86 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:

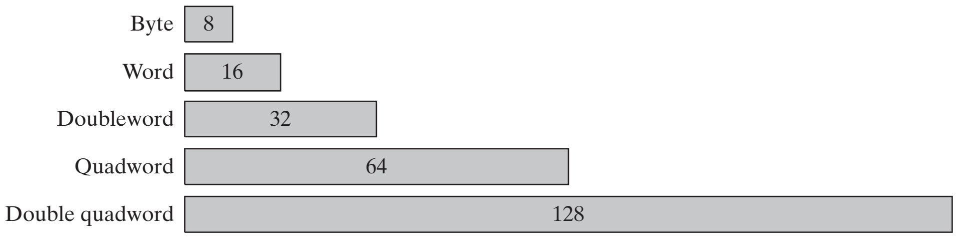


Table 1-4 shows the range of possible values for each type of unsigned integer.

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* (1 MByte) is equal to 2^{20}, or 1,048,576 bytes.
- One *gigabyte* (1 GByte) is equal to 2^{30}, or 1024^3, or 1,073,741,824 bytes.
- One *terabyte* (1 TByte) 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.

Table 1-4 Ranges and Sizes of Unsigned Integer Types.

Type	Range	Storage Size in Bits
Unsigned byte	0 to $2^8 - 1$	8
Unsigned word	0 to $2^{16} - 1$	16
Unsigned doubleword	0 to $2^{32} - 1$	32
Unsigned quadword	0 to $2^{64} - 1$	64
Unsigned double quadword	0 to $2^{128} - 1$	128

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.

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

The following example shows how binary 0001 0110 1010 0111 1001 0100 is equivalent to hexadecimal 16A794:

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

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 integer's decimal value:

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

The formula can be generalized for any *n*-digit hexadecimal integer:

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

In general, you can convert an *n*-digit integer in any base B to decimal using the following formula: $dec = (D_{n-1} \times B^{n-1}) + (D_{n-2} \times B^{n-2}) + \cdots + (D_1 \times B^1) + (D_0 \times B^0)$.

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:

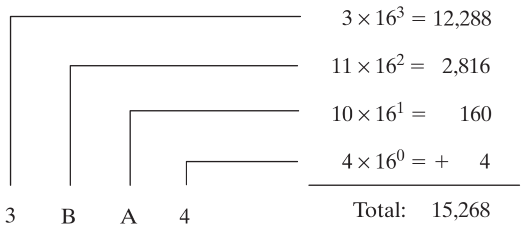


Table 1-6 lists the powers of 16 from 16^0 to 16^7.

Table 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

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:

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

The resulting hexadecimal number is assembled from the digits in the remainder column, starting from the last row and working upward to the top row. In this example, the hexadecimal representation is **1A6**. The same algorithm was used for binary integers in Section 1.3.1. To convert from decimal into some other number base other than hexadecimal, replace the divisor (16) in each calculation with the desired number base.

1.3.5 Hexadecimal Addition

Debugging utility programs (known as *debuggers*) usually display memory addresses in hexadecimal. It is often necessary to add two addresses in order to locate a new address. Fortunately, hexadecimal addition works the same way as decimal addition, if you just change the number base.

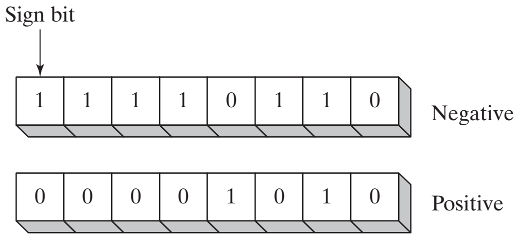
Suppose we want to add two numbers X and Y, using numbering base b. We will number their digits from the lowest position (x_0) to the highest. If we add digits x_i and y_i in X and Y, we produce the value s_i. If $s_i \geq b$, we recalculate $s_i = (s_i \text{ MOD } b)$ and generate a carry value of 1. When we move to the next pair of digits x_{i+1} and y_{i+1}, we add the carry value to their sum.

For example, let's add the hexadecimal values 6A2 and 49A. In the lowest digit position, 2 + A = decimal 12, so there is no carry and we use C to indicate the hexadecimal sum digit. In the next position, A + 9 = decimal 19, so there is a carry because $19 \geq 16$, the number base. We calculate 19 MOD 16 = 3, and carry a 1 into the third digit position. Finally, we add 1 + 6 + 4 = decimal 11, which is shown as the letter B in the third position of the sum. The hexadecimal sum is B3C.

Carry	1		
X	6	A	2
Y	4	9	A
S	B	3	C

1.3.6 Signed Binary Integers

Signed binary integers are positive or negative. For x86 processors, the 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 Representation

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:

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

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.

Hexadecimal Two's Complement To create 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 examples of hexadecimal integers converted to their two's complements:

```
6A3D --> 95C2 + 1 --> 95C3
95C3 --> 6A3C + 1 --> 6A3D
```

Converting Signed Binary to Decimal Use the following algorithm to calculate the decimal equivalent of a signed binary integer:

- If the highest bit is a 1, the number is stored in two's-complement notation. Create 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 create its two's complement, and then 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: Create the two's complement	`00010000`
Step 4: Convert to decimal	`16`

Because the original integer (11110000) was negative, we know that its decimal value is −16.

Converting Signed Decimal to Binary To create 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, create 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 create 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, create 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, create 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 ≥ 8, the number is negative; if the digit is ≤ 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.

Table 1-7 Ranges and Sizes of Signed Integer Types.

Type	Range	Storage Size in Bits
Signed byte	-2^{7} to $+2^{7}-1$	8
Signed word	-2^{15} to $+2^{15}-1$	16
Signed doubleword	-2^{31} to $+2^{31}-1$	32
Signed quadword	-2^{63} to $+2^{63}-1$	64
Signed double quadword	-2^{127} to $+2^{127}-1$	128

1.3.7 Binary Subtraction

Subtracting a smaller unsigned binary number from a large one is easy if you go about it in the same way you handle decimal subtraction. Here's an example:

```
  0 1 1 0 1     (decimal 13)
- 0 0 1 1 1     (decimal 7)
-----------
```

Subtracting the bits in position 0 is straightforward:

```
  0 1 1 0 1
- 0 0 1 1 1
-----------
          0
```

In the next position (0 – 1), we are forced to borrow a 1 from the next position to the left. Here's the result of subtracting 1 from 2:

```
  0 1 0 0 1
- 0 0 1 1 1
-----------
        1 0
```

In the next bit position, we again have to borrow a bit from the column just to the left and subtract 1 from 2:

```
  0 0 0 1 1
- 0 0 1 1 1
-----------
      1 1 0
```

Finally, the two high-order bits are zero minus zero:

```
  0 0 0 1 1
- 0 0 1 1 1
-----------
  0 0 1 1 0     (decimal 6)
```

A simpler way to approach binary subtraction is to reverse the sign of the value being subtracted, and then add the two values. This method requires you to have an extra empty bit to hold the number's sign. Let's try it with the same problem we just calculated: (01101 minus 00111). First, we negate 00111 by inverting its bits (11000) and adding 1, producing 11001. Next, we add the binary values and ignore the carry out of the highest bit:

```
0 1 1 0 1       (+13)
1 1 0 0 1       (-7)
---------
0 0 1 1 0       (+6)
```

The result, +6, is exactly what we expected.

1.3.8 Character Storage

If computers only store binary data, how do they represent characters? They use a *character set*, which is a mapping of characters to integers. In earlier times, character sets used only 8 bits. Even now, 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 through 255 represent graphic symbols and Greek characters.

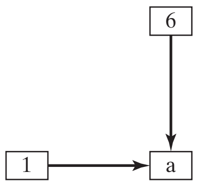
ANSI Character Set The American National Standards Institute (ANSI) defines an 8-bit character set that represents up to 256 characters. The first 128 characters correspond to the letters and symbols on a standard U.S. keyboard. The second 128 characters represent special characters such as letters in international alphabets, accents, currency symbols, and fractions. Early version of Microsoft Windows used the ANSI character set.

Unicode Standard Today, computers must be able to represent a wide variety of international languages in computer software. As a result, the *Unicode* standard was created as a universal way of defining characters and symbols. It defines numeric codes (called *code points*) 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 transformation formats are used to transform code points into displayable characters:

- **UTF-8** is used in HTML, and has the same byte values as ASCII.
- **UTF-16** is used in environments that balance efficient access to characters with economical use of storage. Recent versions of Microsoft Windows, 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.

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 Windows operating system 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 Windows Console 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, and a complete list may be found in the inside front cover of this book.

Table 1-8 ASCII Control Characters.

ASCII Code (Decimal)	Description
8	Backspace (moves one column to the left)
9	Horizontal tab (skips forward *n* columns)
10	Line feed (moves to next output line)
12	Form feed (moves to next printer page)
13	Carriage return (moves to leftmost output column)
27	Escape character

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 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 (such as 8, 16, 32, or 64).
- A *digit string* is a string of ASCII characters, such as "123" or "65." 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 1-9 Types of Digit Strings.

Format	Value
Binary digit string	"01000001"
Decimal digit string	"65"
Hexadecimal digit string	"41"
Octal digit string	"101"

1.4 Boolean Expressions

Boolean algebra defines a set of operations on the values **true** and **false**. It was invented by George Boole, a mid-nineteenth-century mathematician. When early digital computers were invented, it was found that Boole's algebra could be used to describe the design of digital circuits. At the same time, boolean expressions are used in computer programs to express logical operations.

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 $\neg$ or ~ or '
- AND: notated as $\wedge$ or $\bullet$
- OR: notated as $\vee$ 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
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	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 use 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

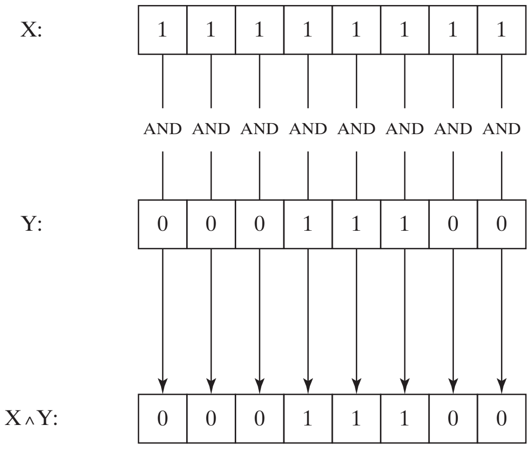
The output is true only when both inputs are true. This corresponds to the logical AND used in compound boolean expressions in C++ and Java.

The AND operation is often carried out at the bit level in assembly language. In the following example, each bit in X is ANDed with its corresponding bit in Y:

```
X:         11111111
Y:         00011100
X ∧ Y:     00011100
```

As Figure 1-2 shows, each bit of the resulting value, 00011100, represents the result of ANDing the corresponding bits in X and Y.

Figure 1–2 ANDing the bits of two binary integers.



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

X	Y	X ∨ Y
F	F	F
F	T	T
T	F	T
T	T	T

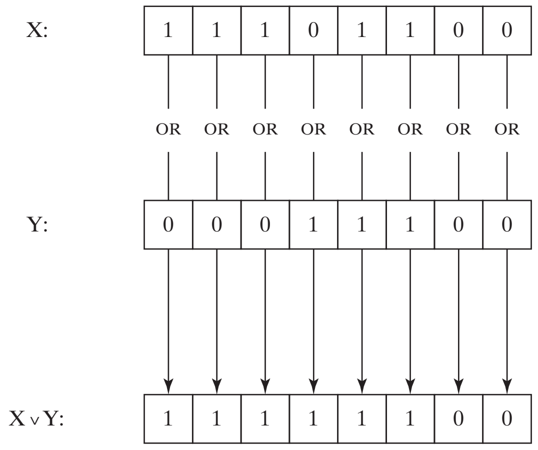
The output is false only when both inputs are false. This truth table corresponds to the logical OR used in compound boolean expressions in C++ and Java.

The OR operation is often carried out at the bit level. In the following example, each bit in X is ORed with its corresponding bit in Y, producing 11111100:

```
X:        11101100
Y:        00011100
X ∨ Y:    11111100
```

As shown in Figure 1-3, the bits are ORed individually, producing a corresponding bit in the result.

FIGURE 1–3 ORing the bits in two binary integers.



Operator Precedence *Operator precedence rules* are used to indicate which operators execute first in expressions involving multiple operators. 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. You can use parentheses to force the initial evaluation of an expression:

Expression	Order of Operations
$\neg X \vee Y$	NOT, then OR
$\neg(X \vee Y)$	OR, then NOT
$X \vee (Y \wedge Z)$	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, 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: $\neg X \vee Y$

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

Example 2: $X \wedge \neg Y$

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

Example 3: $(Y \wedge S) \vee (X \wedge \neg S)$

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

The boolean function in Example 3 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: