Data Representation Chapter One

Probably the biggest stumbling block most beginners encounter when attempting to learn assembly language is the common use of the binary and hexadecimal numbering systems. Many programmers think that hexadecimal (or hex[1]) numbers represent absolute proof that God never intended anyone to work in assembly language. While it is true that hexadecimal numbers are a little different from what you may be used to, their advantages outweigh their disadvantages by a large margin. Nevertheless, understanding these numbering systems is important because their use simplifies other complex topics including boolean algebra and logic design, signed numeric representation, character codes, and packed data.

1.0 Chapter Overview

This chapter discusses several important concepts including the binary and hexadecimal numbering systems, binary data organization (bits, nibbles, bytes, words, and double words), signed and unsigned numbering systems, arithmetic, logical, shift, and rotate operations on binary values, bit fields and packed data, and the ASCII character set. This is basic material and the remainder of this text depends upon your understanding of these concepts. If you are already familiar with these terms from other courses or study, you should at least skim this material before proceeding to the next chapter. If you are unfamiliar with this material, or only vaguely familiar with it, you should study it carefully before proceeding. *All of the material in this chapter is important!* Do not skip over any material.

1.1 Numbering Systems

Most modern computer systems do not represent numeric values using the decimal system. Instead, they typically use a binary or two's complement numbering system. To understand the limitations of computer arithmetic, you must understand how computers represent numbers.

1.1.1 A Review of the Decimal System

You've been using the decimal (base 10) numbering system for so long that you probably take it for granted. When you see a number like "123", you don't think about the value 123; rather, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents:

$$1*10^2 + 2 * 10^1 + 3*10^0$$

or

$$100+20+3$$

Each digit appearing to the left of the decimal point represents a value between zero and nine times an increasing power of ten. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten. For example, the value 123.456 means:

$$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$$

or

1. Hexadecimal is often abbreviated as *hex* even though, technically speaking, hex means base six, not base sixteen.

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

1.1.2 The Binary Numbering System

Most modern computer systems (including the IBM PC) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +5v). With two such levels we can represent exactly two different values. These could be any two different values, but by convention we use the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system. Since there is a correspondence between the logic levels used by the 80x86 and the two digits used in the binary numbering system, it should come as no surprise that the IBM PC employs the binary numbering system.

The binary numbering system works just like the decimal numbering system, with two exceptions: binary only allows the digits 0 and 1 (rather than 0-9), and binary uses powers of two rather than powers of ten. Therefore, it is very easy to convert a binary number to decimal. For each "1" in the binary string, add in 2^n where "n" is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$$
$$=$$
$$128 + 64 + 8 + 2$$
$$=$$
$$202_{10}$$

To convert decimal to binary is slightly more difficult. You must find those powers of two which, when added together, produce the decimal result. The easiest method is to work from the a large power of two down to 2^0. Consider the decimal value 1359:

- $2^{10} = 1024$, $2^{11} = 2048$. So 1024 is the largest power of two less than 1359. Subtract 1024 from 1359 and begin the binary value on the left with a "1" digit. Binary = "1", Decimal result is 1359 - 1024 = 335.
- The next lower power of two ($2^9 = 512$) is greater than the result from above, so add a "0" to the end of the binary string. Binary = "10", Decimal result is still 335.
- The next lower power of two is 256 (2^8). Subtract this from 335 and add a "1" digit to the end of the binary number. Binary = "101", Decimal result is 79.
- 128 (2^7) is greater than 79, so tack a "0" to the end of the binary string. Binary = "1010", Decimal result remains 79.
- The next lower power of two ($2^6 = 64$) is less than 79, so subtract 64 and append a "1" to the end of the binary string. Binary = "10101", Decimal result is 15.
- 15 is less than the next power of two ($2^5 = 32$) so simply add a "0" to the end of the binary string. Binary = "101010", Decimal result is still 15.
- 16 (2^4) is greater than the remainder so far, so append a "0" to the end of the binary string. Binary = "1010100", Decimal result is 15.
- 2^3 (eight) is less than 15, so stick another "1" digit on the end of the binary string. Binary = "10101001", Decimal result is 7.
- 2^2 is less than seven, so subtract four from seven and append another one to the binary string. Binary = "101010011", decimal result is 3.
- 2^1 is less than three, so append a one to the end of the binary string and subtract two from the decimal value. Binary = "1010100111", Decimal result is now 1.
- Finally, the decimal result is one, which is 2^0, so add a final "1" to the end of the binary string. The final binary result is "10101001111"

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs.

1.1.3 Binary Formats

In the purest sense, every binary number contains an infinite number of digits (or *bits* which is short for binary digits). For example, we can represent the number five by:

101 00000101 000000000101 ... 000000000000101

Any number of leading zero bits may precede the binary number without changing its value.

We will adopt the convention ignoring any leading zeros. For example, 101_2 represents the number five. Since the 80x86 works with groups of eight bits, we'll find it much easier to zero extend all binary numbers to some multiple of four or eight bits. Therefore, following this convention, we'd represent the number five as 0101_2 or 00000101_2.

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. We'll adopt a similar convention in this text for binary numbers. We will separate each group of four binary bits with a space. For example, the binary value 1010111110110010 will be written 1010 1111 1011 0010.

We often pack several values together into the same binary number. One form of the 80x86 MOV instruction (see appendix D) uses the binary encoding 1011 0rrr dddd dddd to pack three items into 16 bits: a five-bit operation code (10110), a three-bit register field (rrr), and an eight-bit immediate value (dddd dddd). For convenience, we'll assign a numeric value to each bit position. We'll number each bit as follows:

1) The rightmost bit in a binary number is bit position zero.

2) Each bit to the left is given the next successive bit number.

An eight-bit binary value uses bits zero through seven:

$$X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

A 16-bit binary value uses bit positions zero through fifteen:

$$X_{15}\ X_{14}\ X_{13}\ X_{12}\ X_{11}\ X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

Bit zero is usually referred to as the *low order* (L.O.) bit. The left-most bit is typically called the *high order* (H.O.) bit. We'll refer to the intermediate bits by their respective bit numbers.

1.2 Data Organization

In pure mathematics a value may take an arbitrary number of bits. Computers, on the other hand, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called *nibbles*), groups of eight bits (called *bytes*), groups of 16 bits (called *words*), and more. The sizes are not arbitrary. There is a good reason for these particular values. This section will describe the bit groups commonly used on the Intel 80x86 chips.

1.2.1 Bits

The smallest "unit" of data on a binary computer is a single *bit*. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are *not* limited to representing binary data types (that is, those objects which have only two distinct values). You could use a single bit to represent the numbers 723 and 1,245. Or perhaps 6,254 and 5. You could also use a single bit to represent the colors red and blue. You could even represent two unrelated objects with a single bit,. For example, you could represent the color red and the number 3,256 with a single bit. You can represent *any* two different values with a single bit. However, you can represent *only* two different values with a single bit.

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: *data is what you define it to be*. If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any true meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or blue later.

Since most items you'll be trying to model require more than two different values, single bit values aren't the most popular data type you'll use. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. However, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

1.2.2 Nibbles

A *nibble* is a collection of four bits. It wouldn't be a particularly interesting data structure except for two items: BCD (*binary coded decimal*) numbers and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits (see "The Hexadecimal Numbering System" on page 17). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits. In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

1.2.3 Bytes

Without question, the most important data structure used by the 80x86 microprocessor is the byte. A byte consists of eight bits and is the smallest addressable datum (data item) on the 80x86 microprocessor. Main memory and I/O addresses on the 80x86 are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an eight-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven using the convention in Figure 1.1.

Bit 0 is the *low order bit* or *least significant bit*, bit 7 is the *high order bit* or *most significant bit* of the byte. We'll refer to all other bits by their number.

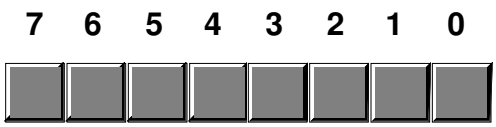

Figure 1.1: Bit Numbering in a Byte

Note that a byte also contains exactly two nibbles (see Figure 1.2).

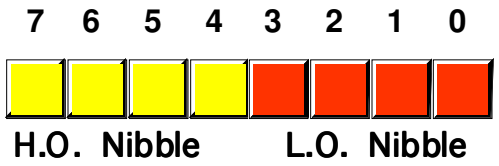

Figure 1.2: The Two Nibbles in a Byte

Bits 0..3 comprise the *low order nibble*, bits 4..7 form the *high order nibble*. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits.

Since a byte contains eight bits, it can represent 2^8, or 256, different values. Generally, we'll use a byte to represent numeric values in the range 0..255, signed numbers in the range -128..+127 (see "Signed and Unsigned Numbers" on page 23), ASCII/IBM character codes, and other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the 80x86 is a byte addressable machine (see "Memory Layout and Access" on page 145), it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we'll often represent the boolean values true and false by 00000001_2 and 00000000_2 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, the IBM PC uses a variant of the ASCII character set (see "The ASCII Character Set" on page 28). There are 128 defined codes in the ASCII character set. IBM uses the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols. See Appendix A for the character/code assignments.

1.2.4 Words

A word is a group of 16 bits. We'll number the bits in a word starting from zero on up to fifteen. The bit numbering appears in Figure 1.3.

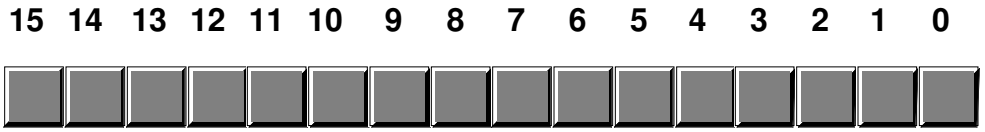

Figure 1.3: Bit Numbers in a Word

Like the byte, bit 0 is the low order bit and bit 15 is the high order bit. When referencing the other bits in a word use their bit position number.

Notice that a word contains exactly two bytes. Bits 0 through 7 form the low order byte, bits 8 through 15 form the high order byte (see Figure 1.4).

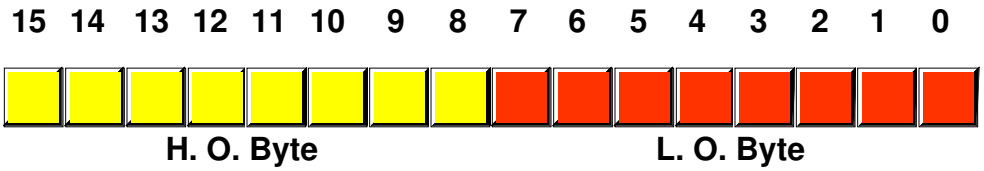

Figure 1.4: The Two Bytes in a Word

Naturally, a word may be further broken down into four nibbles as shown in Figure 1.5.

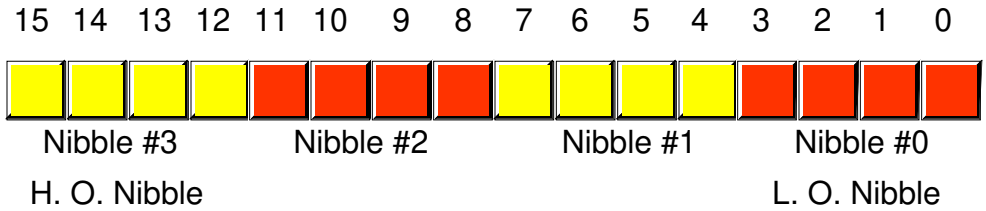

Figure 1.5: Nibbles in a Word

Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. The other two nibbles are "nibble one" or "nibble two".

With 16 bits, you can represent 2^{16} (65,536) different values. These could be the values in the range 0..65,535 (or, as is usually the case, -32,768..+32,767) or any other data type with no more than 65,536 values. The three major uses for words are integer values, offsets, and segment values (see "Memory Layout and Access" on page 145 for a description of segments and offsets).

Words can represent integer values in the range 0..65,535 or -32,768..32,767. Unsigned numeric values are represented by the binary value corresponding to the bits in the word. Signed numeric values use the two's complement form for numeric values (see "Signed and Unsigned Numbers" on page 23). Segment values, which are always 16 bits long, constitute the *paragraph address* of a code, data, extra, or stack segment in memory.

1.2.5 Double Words

A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Figure 1.6.

Figure 1.6: Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, or four different bytes, or eight different nibbles (see Figure 1.7).

Double words can represent all kinds of different things. First and foremost on the list is a segmented address. Another common item represented with a double word is a 32-bit

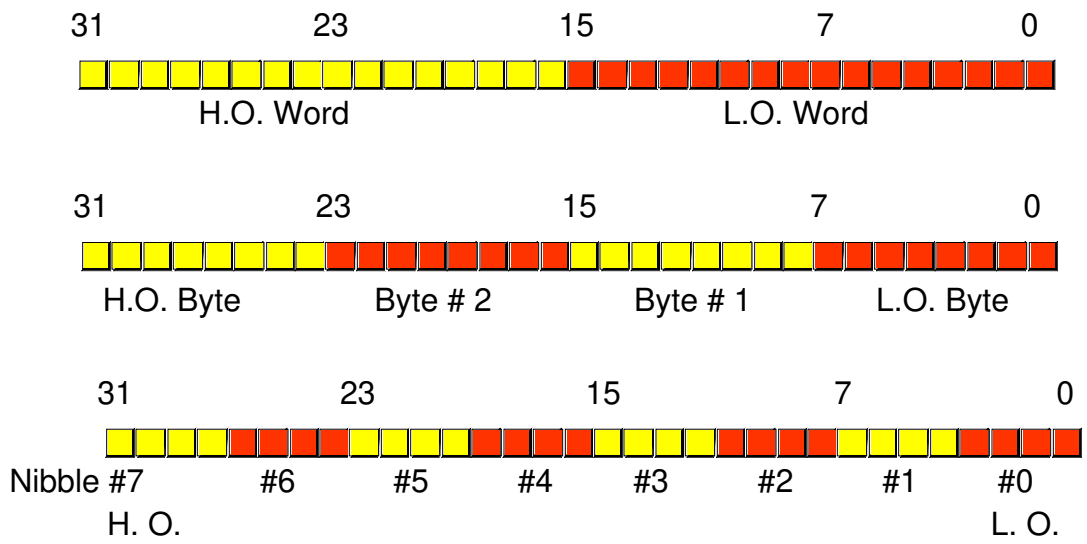

Figure 1.7: Nibbles, Bytes, and Words in a Double Word

integer value (which allows unsigned numbers in the range 0..4,294,967,295 or signed numbers in the range -2,147,483,648..2,147,483,647). 32-bit floating point values also fit into a double word. Most of the time, we'll use double words to hold segmented addresses.

1.3 The Hexadecimal Numbering System

A big problem with the binary system is verbosity. To represent the value 202_{10} requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy. Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we're looking for: they're very compact, and it's simple to convert them to binary and vice versa. Because of this, most binary computer systems today use the hexadecimal numbering system[2]. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

$$1 * 16^3 + 2 * 16^2 + 3 * 16^1 + 4 * 16^0$$

or

$$4096 + 512 + 48 + 4 = 4660_{10}.$$

Each hexadecimal digit can represent one of sixteen values between 0 and 15_{10}. Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 10_{10} through 15_{10}. Rather than create new symbols for these digits, we'll use the letters A through F. The following are all examples of valid hexadecimal numbers:

2. Digital Equipment is the only major holdout. They still use octal numbers in most of their systems. A legacy of the days when they produced 12-bit machines.

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

Since we'll often need to enter hexadecimal numbers into the computer system, we'll need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We'll adopt the following conventions:

- All numeric values (regardless of their radix) begin with a decimal digit.
- All hexadecimal values end with the letter "h", e.g., 123A4h[3].
- All binary values end with the letter "b".
- Decimal numbers *may* have a "t" or "d" suffix.

Examples of valid hexadecimal numbers:

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table:

Table 1: Binary/Hex Conversion

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

This table provides all the information you'll ever need to convert any hexadecimal number into a binary number or vice versa.

To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert

3. Actually, following hexadecimal values with an "h" is an Intel convention, not a general convention. The 68000 and 65c816 assemblers used in the Macintosh and Apple II denote hexadecimal numbers by prefacing the hex value with a "$" symbol.

0ABCDh into a binary value, simply convert each hexadecimal digit according to the table above:

0	A	B	C	D	Hexadecimal
0000	1010	1011	1100	1101	Binary

To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010 1100 1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, e.g., 2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a calculator that will do the conversion for you, you'll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

1.4 Arithmetic Operations on Binary and Hexadecimal Numbers

There are several operations we can perform on binary and hexadecimal numbers. For example, we can add, subtract, multiply, divide, and perform other arithmetic operations. Although you needn't become an expert at it, you should be able to, in a pinch, perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually, the correct way to perform such arithmetic operations is to have a calculator which does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Manufacturers of Hexadecimal Calculators:

- Casio
- Hewlett-Packard
- Sharp
- Texas Instruments

This list is, by no means, exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch . However, they are more expensive than the others. Sharp and Casio produce units which sell for well under $50. If you plan on doing any assembly language programming at all, owning one of these calculators is essential.

Another alternative to purchasing a hexadecimal calculator is to obtain a TSR (Terminate and Stay Resident) program such as SideKick^{tm} which contains a built-in calculator. However, unless you already have one of these programs, or you need some of the other features they offer, such programs are not a particularly good value since they cost more than an actual calculator and are not as convenient to use.

To understand why you should spend the money on a calculator, consider the following arithmetic problem:

```
  9h
+ 1h
----
```

You're probably tempted to write in the answer "10h" as the solution to this problem. But that is not correct! The correct answer is ten, which is "0Ah", not sixteen which is "10h". A similar problem exists with the arithmetic problem:

```
  10h
- 1h
----
```

You're probably tempted to answer "9h" even though the true answer is "0Fh". Remember, this problem is asking "what is the difference between sixteen and one?" The answer, of course, is fifteen which is "0Fh".

Even if the two problems above don't bother you, in a stressful situation your brain will switch back into decimal mode while you're thinking about something else and you'll produce the incorrect result. Moral of the story – if you must do an arithmetic computation using hexadecimal numbers by hand, take your time and be careful about it. Either that, or convert the numbers to decimal, perform the operation in decimal, and convert them back to hexadecimal.

You should never perform binary arithmetic computations. Since binary numbers usually contain long strings of bits, there is too much of an opportunity for you to make a mistake. Always convert binary numbers to hex, perform the operation in hex (preferably with a hex calculator) and convert the result back to binary, if necessary.

1.5 Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. Unlike the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations. It is often easier to do them by hand than to use an electronic device to compute them. The logical AND operation is a dyadic[4] operation (meaning it accepts exactly two operands). These operands are single binary (base 2) bits. The AND operation is:

0 and 0 = 0

0 and 1 = 0

1 and 0 = 0

1 and 1 = 1

A compact way to represent the logical AND operation is with a truth table. A truth table takes the following form:

Table 2: AND Truth Table

AND	0	1
0	0	0
1	0	1

This is just like the multiplication tables you encountered in elementary school. The column on the left and the row at the top represent input values to the AND operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result of logically ANDing those two values together. In English, the logical AND operation is, "If the first operand is one and the second operand is one, the result is one; otherwise the result is zero."

One important fact to note about the logical AND operation is that you can use it to force a zero result. If one of the operands is zero, the result is always zero regardless of the other operand. In the truth table above, for example, the row labelled with a zero input

4. Many texts call this a binary operation. The term dyadic means the same thing and avoids the confusion with the binary numbering system.

contains only zeros and the column labelled with a zero only contains zero results. Conversely, if one operand contains a one, the result is exactly the value of the second operand. These features of the AND operation are very important, particularly when working with bit strings and we want to force individual bits in the string to zero. We will investigate these uses of the logical AND operation in the next section.

The logical OR operation is also a dyadic operation. Its definition is:

0 or 0 = 0

0 or 1 = 1

1 or 0 = 1

1 or 1 = 1

The truth table for the OR operation takes the following form:

Table 3: OR Truth Table

OR	0	1
0	0	1
1	1	1

Colloquially, the logical OR operation is, "If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero." This is also known as the *inclusive-OR* operation.

If one of the operands to the logical-OR operation is a one, the result is always one regardless of the second operand's value. If one operand is zero, the result is always the value of the second operand. Like the logical AND operation, this is an important side-effect of the logical-OR operation that will prove quite useful when working with bit strings (see the next section).

Note that there is a difference between this form of the inclusive logical OR operation and the standard English meaning. Consider the phrase "I am going to the store *or* I am going to the park." Such a statement implies that the speaker is going to the store or to the park but not to both places. Therefore, the English version of logical OR is slightly different than the inclusive-OR operation; indeed, it is closer to the *exclusive-OR* operation.

The logical XOR (exclusive-or) operation is also a dyadic operation. It is defined as follows:

0 xor 0 = 0

0 xor 1 = 1

1 xor 0 = 1

1 xor 1 = 0

The truth table for the XOR operation takes the following form:

Table 4: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English, the logical XOR operation is, "If the first operand or the second operand, but not both, is one, the result is one; otherwise the result is zero." Note that the exclusive-or operation is closer to the English meaning of the word "or" than is the logical OR operation.

If one of the operands to the logical exclusive-OR operation is a one, the result is always the *inverse* of the other operand; that is, if one operand is one, the result is zero if the other operand is one and the result is one if the other operand is zero. If the first operand contains a zero, then the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

The logical NOT operation is a monadic[5] operation (meaning it accepts only one operand). It is:

NOT 0 = 1

NOT 1 = 0

The truth table for the NOT operation takes the following form:

Table 5: NOT Truth Table

NOT	0	1
	1	0

1.6 Logical Operations on Binary Numbers and Bit Strings

As described in the previous section, the logical functions work only with single bit operands. Since the 80x86 uses groups of eight, sixteen, or thirty-two bits, we need to extend the definition of these functions to deal with more than two bits. Logical functions on the 80x86 operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, these functions operate on bit zero producing bit zero of the result. They operate on bit one of the input values producing bit one of the result, etc. For example, if you want to compute the logical AND of the following two eight-bit numbers, you would perform the logical AND operation on each column independently of the others:

```
1011 0101
1110 1110
---------
1010 0100
```

This bit-by-bit form of execution can be easily applied to the other logical operations as well.

Since we've defined logical operations in terms of binary values, you'll find it much easier to perform logical operations on binary values than on values in other bases. Therefore, if you want to perform a logical operation on two hexadecimal numbers, you should convert them to binary first. This applies to most of the basic logical operations on binary numbers (e.g., AND, OR, XOR, etc.).

The ability to force bits to zero or one using the logical AND/OR operations and the ability to invert bits using the logical XOR operation is very important when working with strings of bits (e.g., binary numbers). These operations let you selectively manipulate certain bits within some value while leaving other bits unaffected. For example, if you have an eight-bit binary value 'X' and you want to guarantee that bits four through seven contain zeros, you could logically AND the value 'X' with the binary value 0000 1111. This

5. Monadic means the operator has one operand.

bitwise logical AND operation would force the H.O. four bits to zero and pass the L.O. four bits of 'X' through unchanged. Likewise, you could force the L.O. bit of 'X' to one and invert bit number two of 'X' by logically ORing 'X' with 0000 0001 and logically exclusive-ORing 'X' with 0000 0100, respectively. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is know as *masking* bit strings. We use the term *masking* because we can use certain values (one for AND, zero for OR/XOR) to 'mask out' certain bits from the operation when forcing bits to zero, one, or their inverse.

1.7 Signed and Unsigned Numbers

So far, we've treated binary numbers as unsigned values. The binary number ...00000 represents zero, ...00001 represents one, ...00010 represents two, and so on toward infinity. What about negative numbers? Signed values have been tossed around in previous sections and we've mentioned the two's complement numbering system, but we haven't discussed how to represent negative numbers using the binary numbering system. That is what this section is all about!

To represent signed numbers using the binary numbering system we have to place a restriction on our numbers: they must have a finite and fixed number of bits. As far as the 80x86 goes, this isn't too much of a restriction, after all, the 80x86 can only address a finite number of bits. For our purposes, we're going to severely limit the number of bits to eight, 16, 32, or some other small number of bits.

With a fixed number of bits we can only represent a certain number of objects. For example, with eight bits we can only represent 256 different objects. Negative values are objects in their own right, just like positive numbers. Therefore, we'll have to use some of the 256 different values to represent negative numbers. In other words, we've got to use up some of the positive numbers to represent negative numbers. To make things fair, we'll assign half of the possible combinations to the negative values and half to the positive values. So we can represent the negative values -128..-1 and the positive values 0..127 with a single eight bit byte[6]. With a 16-bit word we can represent values in the range -32,768..+32,767. With a 32-bit double word we can represent values in the range -2,147,483,648..+2,147,483,647. In general, with *n* bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1}-1$.

Okay, so we can represent negative values. Exactly how do we do it? Well, there are many ways, but the 80x86 microprocessor uses the two's complement notation. In the two's complement system, the H.O. bit of a number is a *sign bit*. If the H.O. bit is zero, the number is positive; if the H.O. bit is one, the number is negative. Examples:

For 16-bit numbers:

8000h is negative because the H.O. bit is one.

100h is positive because the H.O. bit is zero.

7FFFh is positive.

0FFFFh is negative.

0FFFh is positive.

If the H.O. bit is zero, then the number is positive and is stored as a standard binary value. If the H.O. bit is one, then the number is negative and is stored in the two's complement form. To convert a positive number to its negative, two's complement form, you use the following algorithm:

1) Invert all the bits in the number, i.e., apply the logical NOT function.

6. Technically, zero is neither positive nor negative. For technical reasons (due to the hardware involved), we'll lump zero in with the positive numbers.

2) Add one to the inverted result.

For example, to compute the eight bit equivalent of -5:

```
0000 0101           Five (in binary).
1111 1010           Invert all the bits.
1111 1011           Add one to obtain result.
```

If we take minus five and perform the two's complement operation on it, we get our original value, 00000101, back again, just as we expect:

```
1111 1011           Two's complement for -5.
0000 0100           Invert all the bits.
0000 0101           Add one to obtain result (+5).
```

The following examples provide some positive and negative 16-bit signed values:

7FFFh: +32767, the largest 16-bit positive number.

8000h: -32768, the smallest 16-bit negative number.

4000h: +16,384.

To convert the numbers above to their negative counterpart (i.e., to negate them), do the following:

```
7FFFh:      0111 1111 1111 1111         +32,767t
            1000 0000 0000 0000         Invert all the bits (8000h)
            1000 0000 0000 0001         Add one (8001h or -32,767t)

8000h:      1000 0000 0000 0000         -32,768t
            0111 1111 1111 1111         Invert all the bits (7FFFh)
            1000 0000 0000 0000         Add one (8000h or -32768t)

4000h:      0100 0000 0000 0000         16,384t
            1011 1111 1111 1111         Invert all the bits (BFFFh)
            1100 0000 0000 0000         Add one (0C000h or -16,384t)
```

8000h inverted becomes 7FFFh. After adding one we obtain 8000h! Wait, what's going on here? -(-32,768) is -32,768? Of course not. But the value +32,768 cannot be represented with a 16-bit signed number, so we cannot negate the smallest negative value. If you attempt this operation, the 80x86 microprocessor will complain about signed arithmetic overflow.

Why bother with such a miserable numbering system? Why not use the H.O. bit as a sign flag, storing the positive equivalent of the number in the remaining bits? The answer lies in the hardware. As it turns out, negating values is the only tedious job. With the two's complement system, most other operations are as easy as the binary system. For example, suppose you were to perform the addition 5+(-5). The result is zero. Consider what happens when we add these two values in the two's complement system:

```
  00000101
  11111011
  --------
1 00000000
```

We end up with a carry into the ninth bit and all other bits are zero. As it turns out, if we ignore the carry out of the H.O. bit, adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. This wouldn't be the case with some other numbering systems.

Except for the questions at the end of this chapter, you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction, NEG (negate), which performs this operation for you. Furthermore, all the hexadecimal

calculators will perform this operation by pressing the change sign key (+/- or CHS). Nevertheless, performing a two's complement by hand is easy, and you should know how to do it.

Once again, you should note that the data represented by a set of binary bits depends entirely on the context. The eight bit binary value 11000000b could represent an IBM/ASCII character, it could represent the unsigned decimal value 192, or it could represent the signed decimal value -64, etc. As the programmer, it is your responsibility to use this data consistently.

1.8 Sign and Zero Extension

Since two's complement format integers have a fixed length, a small problem develops. What happens if you need to convert an eight bit two's complement value to 16 bits? This problem, and its converse (converting a 16 bit value to eight bits) can be accomplished via *sign extension* and *contraction* operations. Likewise, the 80x86 works with fixed length values, even when processing unsigned binary numbers. *Zero extension* lets you convert small unsigned values to larger unsigned values.

Consider the value "-64". The eight bit two's complement value for this number is 0C0h. The 16-bit equivalent of this number is 0FFC0h. Now consider the value "+64". The eight and 16 bit versions of this value are 40h and 0040h. The difference between the eight and 16 bit numbers can be described by the rule: "If the number is negative, the H.O. byte of the 16 bit number contains 0FFh; if the number is positive, the H.O. byte of the 16 bit quantity is zero."

To sign extend a value from some number of bits to a greater number of bits is easy, just copy the sign bit into all the additional bits in the new format. For example, to sign extend an eight bit number to a 16 bit number, simply copy bit seven of the eight bit number into bits 8..15 of the 16 bit number. To sign extend a 16 bit number to a double word, simply copy bit 15 into bits 16..31 of the double word.

Sign extension is required when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division, in particular) may require a sign extension to 32-bits. You must not sign extend unsigned values.

Examples of sign extension:

```
Eight Bits     Sixteen Bits          Thirty-two Bits

 80h            FF80h                 FFFFFF80h
 28h            0028h                 00000028h
 9Ah            FF9Ah                 FFFFFF9Ah
 7Fh            007Fh                 0000007Fh
 ---            1020h                 00001020h
 ---            8088h                 FFFF8088h
```

To extend an unsigned byte you must zero extend the value. Zero extension is very easy – just store a zero into the H.O. byte(s) of the smaller operand. For example, to zero extend the value 82h to 16-bits you simply add a zero to the H.O. byte yielding 0082h.

```
Eight Bits     Sixteen Bits          Thirty-two Bits

 80h            0080h                 00000080h
 28h            0028h                 00000028h
 9Ah            009Ah                 0000009Ah
 7Fh            007Fh                 0000007Fh
 ---            1020h                 00001020h
 ---            8088h                 00008088h
```

Sign contraction, converting a value with some number of bits to the identical value with a fewer number of bits, is a little more troublesome. Sign extension never fails. Given an *m*-bit signed value you can always convert it to an *n*-bit number (where $n > m$) using

sign extension. Unfortunately, given an *n*-bit number, you cannot always convert it to an *m*-bit number if *m* < *n*. For example, consider the value -448. As a 16-bit hexadecimal number, its representation is 0FE40h. Unfortunately, the magnitude of this number is too great to fit into an eight bit value, so you cannot sign contract it to eight bits. This is an example of an overflow condition that occurs upon conversion.

To properly sign contract one value to another, you must look at the H.O. byte(s) that you want to discard. The H.O. bytes you wish to remove must all contain either zero or 0FFh. If you encounter any other values, you cannot contract it without overflow. Finally, the H.O. bit of your resulting value must match *every* bit you've removed from the number. Examples (16 bits to eight bits):

```
FF80h can be sign contracted to 80h
0040h can be sign contracted to 40h
FE40h cannot be sign contracted to 8 bits.
0100h cannot be sign contracted to 8 bits.
```

1.9 Shifts and Rotates

Another set of logical operations which apply to bit strings are the *shift* and *rotate* operations. These two categories can be further broken down into *left shifts, left rotates, right shifts,* and *right rotates*. These operations turn out to be extremely useful to assembly language programmers.

The left shift operation moves each bit in a bit string one position to the left (see Figure 1.8).

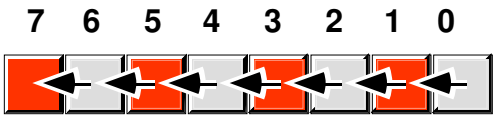

Figure 1.8: Shift Left Operation

Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, etc. There are, of course, two questions that naturally arise: "What goes into bit zero?" and "Where does bit seven wind up?" Well, that depends on the context. We'll shift the value zero into the L.O. bit, and the previous value of bit seven will be the *carry* out of this operation.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a zero to the right of the number) effectively multiplies it by ten (the radix):

```
1234 SHL 1 = 12340        (SHL 1 = shift left one position)
```

Since the radix of a binary number is two, shifting it left multiplies it by two. If you shift a binary value to the left twice, you multiply it by two twice (i.e., you multiply it by four). If you shift a binary value to the left three times, you multiply it by eight (2*2*2). In general, if you shift a value to the left *n* times, you multiply that value by 2^n.

A right shift operation works the same way, except we're moving the data in the opposite direction. Bit seven moves into bit six, bit six moves into bit five, bit five moves into bit four, etc. During a right shift, we'll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 1.9).

Since a left shift is equivalent to a multiplication by two, it should come as no surprise that a right shift is roughly comparable to a division by two (or, in general, a division by the radix of the number). If you perform *n* right shifts, you will divide that number by 2^n.

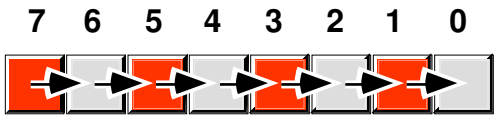

Figure 1.9: Shift Right Operation

There is one problem with shift rights with respect to division: as described above a shift right is only equivalent to an *unsigned* division by two. For example, if you shift the unsigned representation of 254 (0FEh) one place to the right, you get 127 (07Fh), exactly what you would expect. However, if you shift the binary representation of -2 (0FEh) to the right one position, you get 127 (07Fh), which is *not* correct. This problem occurs because we're shifting a zero into bit seven. If bit seven previously contained a one, we're changing it from a negative to a positive number. Not a good thing when dividing by two.

To use the shift right as a division operator, we must define a third shift operation: *arithmetic shift right*[7]. An arithmetic shift right works just like the normal shift right operation (a *logical shift right*) with one exception: instead of shifting a zero into bit seven, an arithmetic shift right operation leaves bit seven alone, that is, during the shift operation it does not modify the value of bit seven as Figure 1.10 shows.

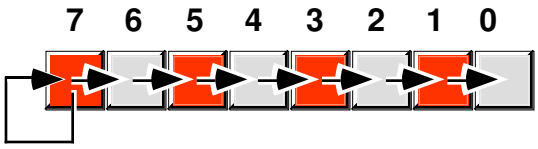

Figure 1.10: Arithmetic Shift Right Operation

This generally produces the result you expect. For example, if you perform the arithmetic shift right operation on -2 (0FEh) you get -1 (0FFh). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer *which is less than or equal to the actual result*. Based on experiences with high level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates towards zero. But this simply isn't the case. For example, if you apply the arithmetic shift right operation on -1 (0FFh), the result is -1, not zero. -1 is less than zero so the arithmetic shift right operation rounds towards minus one. This is not a "bug" in the arithmetic shift right operation. This is the way integer division typically gets defined. The 80x86 integer division instruction also produces this result.

Another pair of useful operations are *rotate left* and *rotate right*. These operations behave like the shift left and shift right operations with one major difference: the bit shifted out from one end is shifted back in at the other end.

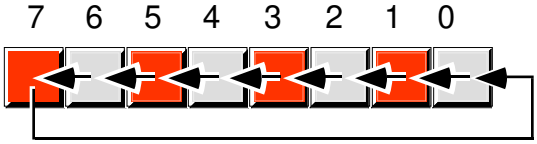

Figure 1.11: Rotate Left Operation

7. There is no need for an arithmetic shift left. The standard shift left operation works for both signed and unsigned numbers, assuming no overflow occurs.

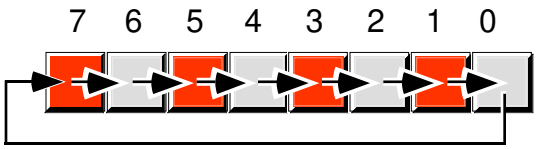

Figure 1.12: Rotate Right Operation

1.10 Bit Fields and Packed Data

Although the 80x86 operates most efficiently on byte, word, and double word data types, occasionally you'll need to work with a data type that uses some number of bits other than eight, 16, or 32. For example, consider a date of the form "4/2/88". It takes three numeric values to represent this date: a month, day, and year value. Months, of course, take on the values 1..12. It will require at least four bits (maximum of sixteen different values) to represent the month. Days range between 1..31. So it will take five bits (maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires seven bits (which can be used to represent up to 128 different values). Four plus five plus seven is 16 bits, or two bytes. In other words, we can pack our date data into two bytes rather than the three that would be required if we used a separate byte for each of the month, day, and year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store a lot of dates. The bits could be arranged as shown in .

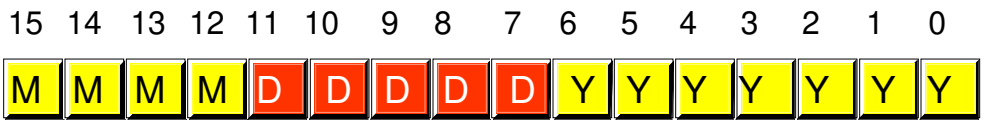

Figure 1.13: Packed Date Format

MMMM represents the four bits making up the month value, DDDDD represents the five bits making up the day, and YYYYYYY is the seven bits comprising the year. Each collection of bits representing a data item is a *bit field*. April 2nd, 1988 would be represented as 4158h:

```
0100 00010 1011000    = 0100 0001 0101 1000b or 4158h
 4     2      88
```

Although packed values are *space efficient* (that is, very efficient in terms of memory usage), they are computationally *inefficient* (slow!). The reason? It takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything.

Examples of practical packed data types abound. You could pack eight boolean values into a single byte, you could pack two BCD digits into a byte, etc.

1.11 The ASCII Character Set

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through

1Fh (31), form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer/display control operations rather than displaying symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters[8], line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code 20h) and the numeric digits (ASCII codes 30h..39h). Note that the numeric digits differ from their numeric values only in the H.O. nibble. By subtracting 30h from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters is reserved for the upper case alphabetic characters. The ASCII codes for the characters "A"..."Z" lie in the range 41h..5Ah (65..90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes are reserved for the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes 61h..7Ah. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position. For example, consider the character code for "E" and "e" in Figure 1.14.

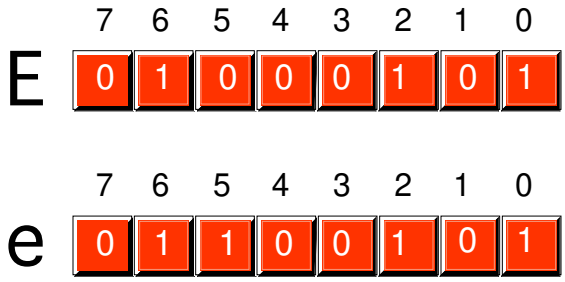

Figure 1.14: ASCII Codes for "E" and "e".

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

Indeed, bits five and six determine which of the four groups in the ASCII character set you're in:

8. Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left hand side of the paper.

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

So you could, for instance, convert any upper or lower case (or corresponding special) character to its equivalent control character by setting bits five and six to zero.

Consider, for a moment, the ASCII codes of the numeric digit characters:

Char Dec Hex

"0"	48	30h
"1"	49	31h
"2"	50	32h
"3"	51	33h
"4"	52	34h
"5"	53	35h
"6"	54	36h
"7"	55	37h
"8"	56	38h
"9"	57	39h

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important – the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to zero) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to three. Note that you can use the logical-AND operation to force the H.O. bits to zero; likewise, you can use the logical-OR operation to force the H.O. bits to 0011 (three).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 (31h 32h 33h) in this fashion yields three bytes: 010203h, not the correct value which is 7Bh. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

Bit seven in standard ASCII is always zero. This means that the ASCII character set consumes only half of the possible character codes in an eight bit byte. IBM uses the remaining 128 character codes for various special characters including international characters (those with accents, etc.), math symbols, and line drawing characters. Note that these extra characters are a non-standard extension to the ASCII character set. Of course, the name IBM has considerable clout, so almost all modern personal computers based on the 80x86 with a video display support the extended IBM/ASCII character set. Most printers support IBM's character set as well.

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set; however you may obtain a PC font which lets you display the extended character set. Other machines (e.g., Amiga and Atari ST) have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a "standard", simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it's true that an "A" on one machine is most likely an "A" on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported – backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, Apple II, and many other systems mark the end of line by a single CR character. UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., "0", "A", "a", etc.).

1.12 Summary

Most modern computer systems use the binary numbering system to represent values. Since binary values are somewhat unwieldy, we'll often use the hexadecimal representation for those values. This is because it is very easy to convert between hexadecimal and binary, unlike the conversion between the more familiar decimal and binary systems. A single hexadecimal digit consumes four binary digits (bits), and we call a group of four bits a nibble. See:

- "The Binary Numbering System" on page 12
- "Binary Formats" on page 13
- "The Hexadecimal Numbering System" on page 17

The 80x86 works best with groups of bits which are eight, 16, or 32 bits long. We call objects of these sizes bytes, words, and double words, respectively. With a byte, we can represent any one of 256 unique values. With a word we can represent one of 65,536 different values. With a double word we can represent over four billion different values. Often we simply represent integer values (signed or unsigned) with bytes, words, and double words; however we'll often represent other quantities as well. See:

- "Data Organization" on page 13
- "Bytes" on page 14
- "Words" on page 15
- "Double Words" on page 16

In order to talk about specific bits within a nibble, byte, word, double word, or other structure, we'll number the bits starting at zero (for the least significant bit) on up to *n*-1

(where *n* is the number of bits in the object). We'll also number nibbles, bytes, and words in large structures in a similar fashion. See:

- "Binary Formats" on page 13

There are many operations we can perform on binary values including normal arithmetic (+, -, *, and /) and the logical operations (AND, OR, XOR, NOT, Shift Left, Shift Right, Rotate Left, and Rotate Right). Logical AND, OR, XOR, and NOT are typically defined for single bit operations. We can extend these to *n* bits by performing bitwise operations. The shifts and rotates are always defined for a fixed length string of bits. See:

- "Arithmetic Operations on Binary and Hexadecimal Numbers" on page 19
- "Logical Operations on Bits" on page 20
- "Logical Operations on Binary Numbers and Bit Strings" on page 22
- "Shifts and Rotates" on page 26

There are two types of integer values which we can represent with binary strings on the 80x86: unsigned integers and signed integers. The 80x86 represents unsigned integers using the standard binary format. It represents signed integers using the two's complement format. While unsigned integers may be of arbitrary length, it only makes sense to talk about fixed length signed binary values. See:

- "Signed and Unsigned Numbers" on page 23
- "Sign and Zero Extension" on page 25

Often it may not be particularly practical to store data in groups of eight, 16, or 32 bits. To conserve space you may want to pack various pieces of data into the same byte, word, or double word. This reduces storage requirements at the expense of having to perform extra operations to pack and unpack the data. See:

- "Bit Fields and Packed Data" on page 28

Character data is probably the most common data type encountered besides integer values. The IBM PC and compatibles use a variant of the ASCII character set – the extended IBM/ASCII character set. The first 128 of these characters are the standard ASCII characters, 128 are special characters created by IBM for international languages, mathematics, and line drawing. Since the use of the ASCII character set is so common in modern programs, familiarity with this character set is essential. See:

- "The ASCII Character Set" on page 28