Data Representation Chapter Three

A major stumbling block many 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.

3.1 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. 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. In addition to the basic material, this chapter also introduces some new HLA statements and HLA Standard Library routines.

3.2 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.

3.2.1 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

In the positional numbering system, 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

100 + 20 + 3 + 0.4 + 0.05 + 0.006

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

3.2.2 The Binary Numbering System

Most modern computer systems (including PCs) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +2.4..5v). With two such levels we can represent exactly two different values. These could be any two different values, but they typically represent 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 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. One method is to work from 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 than79, 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"

If you actually have to convert a decimal number to binary by hand, the algorithm above probably isn't the easiest to master. A simpler solution is the "even/odd – divide by two" algorithm. This algorithm uses the following steps:

- If the number is even, emit a zero. If the number is odd, emit a one.
- Divide the number by two and throw away any fractional component or remainder.

- If the quotient is zero, the algorithm is complete.
- If the quotient is not zero and is odd, insert a one before the current string; if the number is even, prefix your binary string with zero.
- Go back to step two above and repeat.

Fortunately, you'll rarely need to convert decimal numbers directly to binary strings, so neither of these algorithms is particularly important in real life.

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs (even if you don't convert between decimal and binary). So you should be somewhat comfortable with them.

3.2.3 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 0000000000101 ... 000000000000101

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

We will adopt the convention of ignoring any leading zeros if present in a value. For example, 101_2 represents the number five but 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 an underscore. For example, we will write the binary value 1010111110110010 as 1010_1111_1011_0010.

We often pack several values together into the same binary number. One form of the 80x86 MOV instruction uses the binary encoding 1011 0rrr dddd dddd to pack three items into 16 bits: a five-bit operation code (1_0110), 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:

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

A 32-bit binary value uses bit positions zero through 31, etc.

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

3.3 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 (*bytes*), groups of 16 bits (*words*), groups of 32 bits (double words or *dwords*), groups of 64-bits (quad words or *qwords*), 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.

3.3.1 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 real 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.

3.3.2 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[2] 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 since there are 16 unique combinations of a string of four bits:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

2. Binary coded decimal is a numeric scheme used to represent decimal numbers using four bits for each decimal digit.


```
1001
1010
1011
1100
1101
1110
1111
```

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 60). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits (since you can only represent eight different values with three 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.

3.3.3 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 as shown in Figure 3.1.

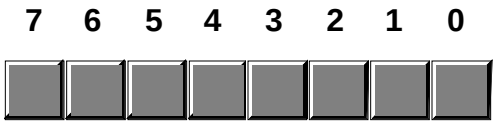

Figure 3.1 Bit Numbering

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.

Note that a byte also contains exactly two nibbles (see Figure 3.2).

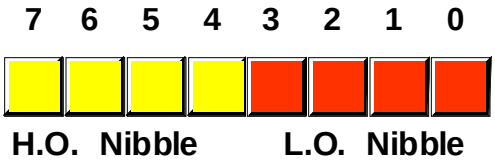

Figure 3.2 The Two Nibbles in a Byte

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 69), 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 the next volume), 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, PCs use a variant of the ASCII character set (see "The ASCII Character Encoding" on page 97). There are 128 defined codes in the ASCII character set. PCs typically use the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols.

Because bytes are the smallest unit of storage in the 80x86 memory space, bytes also happen to be the smallest variable you can create in an HLA program. As you saw in the last chapter, you can declare an eight-bit signed integer variable using the *int8* data type. Since *int8* objects are signed, you can represent values in the range -128..+127 using an *int8* variable (see "Signed and Unsigned Numbers" on page 69 for a discussion of signed number formats). You should only store signed values into *int8* variables; if you want to create an arbitrary byte variable, you should use the *byte* data type, as follows:

```
static
       byteVar: byte;
```

The *byte* data type is a partially untyped data type. The only type information associated with *byte* objects is their size (one byte). You may store any one-byte object (small signed integers, small unsigned integers, characters, etc.) into a byte variable. It is up to you to keep track of the type of object you've put into a byte variable.

3.3.4 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 3.3.

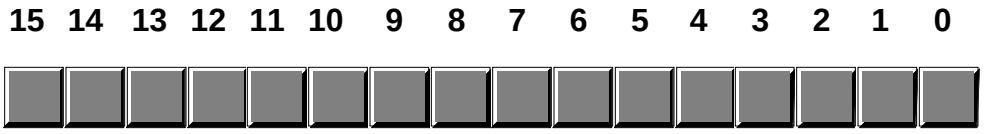

Figure 3.3 Bit Numbers in a Word

Like the byte, bit 0 is the low order bit. For words, 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 3.4).

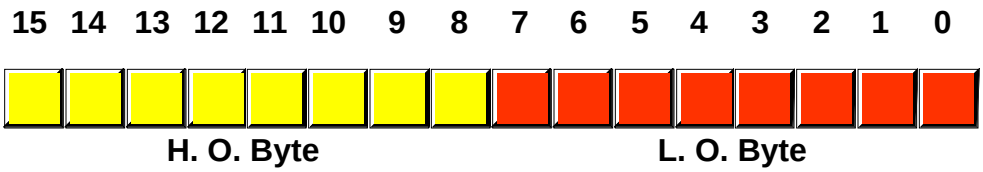

Figure 3.4 The Two Bytes in a Word

Naturally, a word may be further broken down into four nibbles as shown in Figure 3.5.

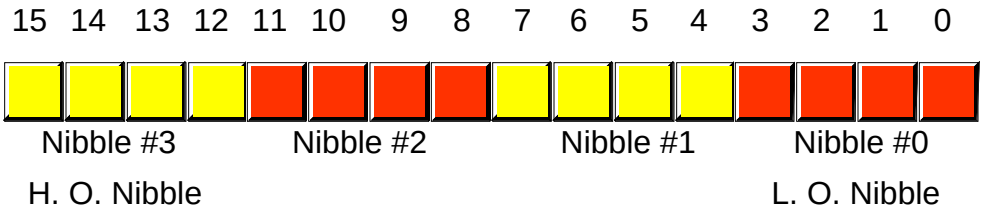

Figure 3.5 Nibbles in a Word

Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. We'll simply refer to the other two nibbles as "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 signed integer values, unsigned integer values, and UNICODE characters.

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 69). As UNICODE characters, words can represent up to 65,536 different characters, allowing the use of non-Roman character sets in a computer program. UNICODE is an international standard, like ASCII, that allows computers to process non-Roman characters like Asian, Greek, and Russian characters.

Like bytes, you can also create word variables in an HLA program. Of course, in the last chapter you saw how to create sixteen-bit signed integer variables using the *int16* data type. To create an arbitrary word variable, just use the *word* data type, as follows:

```
static
      w: word;
```

3.3.5 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 3.6.

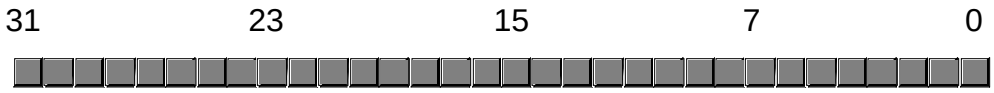

Figure 3.6 Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, four different bytes, or eight different nibbles (see Figure 3.7).

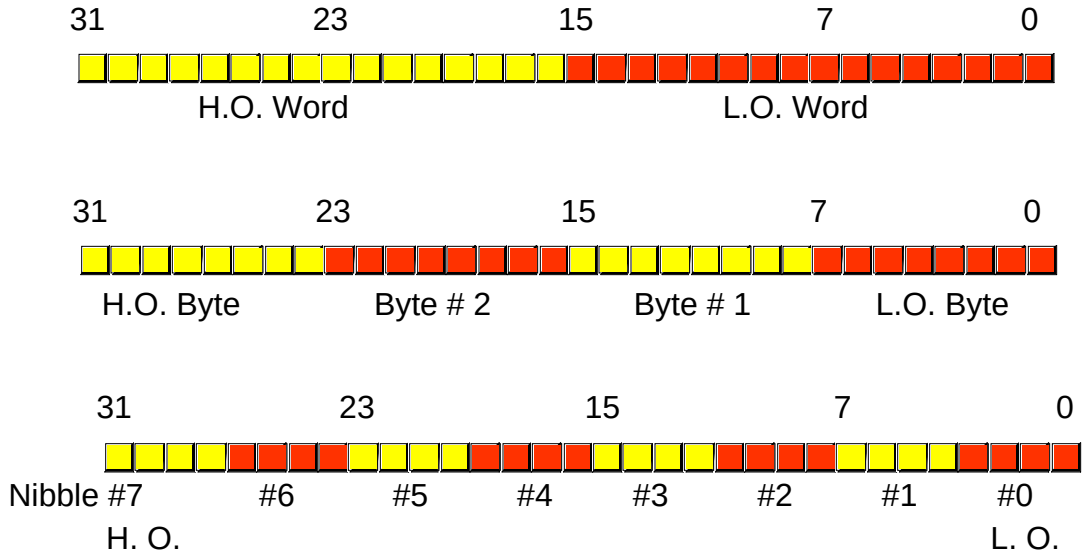

Figure 3.7 Nibbles, Bytes, and Words in a Double Word

Double words can represent all kinds of different things. A common item you will represent with a double word is a 32-bit 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. Another common use for dword objects is to store pointer variables.

In the previous chapter, you saw how to create 32-bit (dword) signed integer variables using the *int32* data type. You can also create an arbitrary double word variable using the *dword* data type as the following example demonstrates:

```
static
        d: dword;
```

3.4 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 computer systems engineers use the hexadecimal numbering system. 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:

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 hexadecimal values begin with a "$" character, e.g., $123A4.
- All binary values begin with a percent sign ("%").
- Decimal numbers do not have a prefix character.
- If the radix is clear from the context, this text may drop the leading "$" or "%" character.

Examples of valid hexadecimal numbers:

$1234 $DEAD $BEEF $AFB $FEED $DEAF

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 4: 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

Table 4: Binary/Hex Conversion

Binary	Hexadecimal
%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 $ABCD 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, i.e., $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.

3.5 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 that does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Some manufacturers of Hexadecimal Calculators (circa 2002):

- 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.

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

```
  $9
+ $1
----
```

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

```
 $10
- $1
----
```

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

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.

3.6 A Note About Numbers vs. Representation

Many people confuse numbers and their representation. A common question beginning assembly language students have is "I've got a binary number in the EAX register, how do I convert that to a hexadecimal number in the EAX register?" The answer is "you don't." Although a strong argument could be made that numbers in memory or in registers are represented in binary, it's best to view values in memory or in a register as *abstract numeric quantities*. Strings of symbols like 128, $80, or %1000_0000 are not different numbers; they are simply different representations for the same abstract quantity that we often refer to as "one hundred twenty-eight." Inside the computer, a number is a number regardless of representation; the only time representation matters is when you input or output the value in a human readable form.

Human readable forms of numeric quantities are always strings of characters. To print the value 128 in human readable form, you must convert the numeric value 128 to the three-character sequence '1' followed by '2' followed by '8'. This would provide the decimal representation of the numeric quantity. If you prefer, you could convert the numeric value 128 to the three character sequence "$80". It's the same number, but we've converted it to a different sequence of characters because (presumably) we wanted to view the number using hexadecimal representation rather than decimal. Likewise, if we want to see the number in binary, then we must convert this numeric value to a string containing a one followed by seven zeros.

By default, HLA displays all byte, word, and dword variables using the hexadecimal numbering system when you use the *stdout.put* routine. Likewise, HLA's *stdout.put* routine will display all register values in hex. Consider the following program that converts values input as decimal numbers to their hexadecimal equivalents:

```
program ConvertToHex;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToHex;
```



```
    stdout.put( "Input a decimal value:" );
    stdin.get( value );
    mov( value, eax );
    stdout.put( "The value ", value, " converted to hex is $", eax, nl );

end ConvertToHex;
```

Program 3.11 Decimal to Hexadecimal Conversion Program

In a similar fashion, the default input base is also hexadecimal for registers and byte, word, or dword variables. The following program is the converse of the one above- it inputs a hexadecimal value and outputs it as decimal:

```
program ConvertToDecimal;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToDecimal;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );  mov( ebx, value );
    stdout.put( "The value $", ebx, " converted to decimal is ", value, nl );

end ConvertToDecimal;
```

Program 3.12 Hexadecimal to Decimal Conversion Program

Just because the HLA *stdout.put* routine chooses decimal as the default output base for *int8*, *int16*, and *int32* variables doesn't mean that these variables hold "decimal" numbers. Remember, memory and registers hold numeric values, not hexadecimal or decimal values. The *stdout.put* routine converts these numeric values to strings and prints the resulting strings. The choice of hexadecimal vs. decimal output was a design choice in the HLA language, nothing more. You could very easily modify HLA so that it outputs registers and *byte*, *word*, or *dword* variables as decimal values rather than as hexadecimal. If you need to print the value of a register or *byte*, *word*, or *dword* variable as a decimal value, simply call one of the putiX routines to do this. The *stdout.puti8* routine will output its parameter as an eight-bit signed integer. Any eight-bit parameter will work. So you could pass an eight-bit register, an *int8* variable, or a *byte* variable as the parameter to *stdout.puti8* and the result will always be decimal. The *stdout.puti16* and *stdout.puti32* provide the same capabilities for 16-bit and 32-bit objects. The following program demonstrates the decimal conversion program (Program 3.12 above) using only the EAX register (i.e., it does not use the variable *iValue*):

```
program ConvertToDecimal2;
#include( "stdlib.hhf" );
begin ConvertToDecimal2;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );
    stdout.put( "The value $", ebx, " converted to decimal is " );
    stdout.puti32( ebx );
    stdout.newln();
```



```
end ConvertToDecimal2;
```

Program 3.13 Variable-less Hexadecimal to Decimal Converter

Note that HLA's *stdin.get* routine uses the same default base for input as *stdout.put* uses for output. That is, if you attempt to read an *int8*, *int16*, or *int32* variable, the default input base is decimal. If you attempt to read a register or *byte*, *word*, or *dword* variable, the default input base is hexadecimal. If you want to change the default input base to decimal when reading a register or a *byte*, *word*, or *dword* variable, then you can use *stdin.geti8*, *stdin.geti16*, or *stdin.geti32*.

If you want to go in the opposite direction, that is you want to input or output an *int8*, *int16*, or *int32* variable as a hexadecimal value, you can call the *stdout.putb*, *stdout.putw*, *stdout.putd*, *stdin.getb*, *stdin.getw*, or *stdin.getd* routines. The *stdout.putb*, *stdout.putw*, and *stdout.putd* routines write eight-bit, 16-bit, or 32-bit objects as hexadecimal values. The *stdin.getb*, *stdin.getw*, and *stdin.getd* routines read eight-bit, 16-bit, and 32-bit values respectively; they return their results in the AL, AX, or EAX registers. The following program demonstrates the use of a few of these routines:

```
program HexIO;

#include( "stdlib.hhf" );

static
    i32: int32;

begin HexIO;

    stdout.put( "Enter a hexadecimal value: " );
    stdin.getd();
    mov( eax, i32 );
    stdout.put( "The value you entered was $" );
    stdout.putd( i32 );
    stdout.newln();

end HexIO;
```

Program 3.14 Demonstration of stdin.getd and stdout.putd

3.7 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[3] 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 5: 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 values in the left column correspond to the leftmost operand of the AND operation. The values in the top row correspond to the rightmost operand of 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." We could also state this as "If either or both operands are zero, 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 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 we want to force individual bits in a bit 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

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

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

Table 6: 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 since it lets you force individual bits to one.

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 7: 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 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 8: NOT Truth Table

NOT	0	1
	1	0

3.8 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 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 known 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' or 'mask in' certain bits from the operation when forcing bits to zero, one, or their inverse.

The 80x86 CPUs support four instructions that apply these bitwise logical operations to their operands. The instructions are AND, OR, XOR, and NOT. The AND, OR, and XOR instructions use the same syntax as the ADD and SUB instructions, that is,

```
and( source, dest );
 or( source, dest );
xor( source, dest );
```

These operands have the same limitations as the ADD operands. Specifically, the *source* operand has to be a constant, memory, or register operand and the *dest* operand must be a memory or register operand. Also, the operands must be the same size and they cannot both be memory operands. These instructions compute the obvious bitwise logical operation via the equation:

dest = *dest* operator *source*

The 80x86 logical NOT instruction, since it has only a single operand, uses a slightly different syntax. This instruction takes the following form:

```
not( dest );
```

Note that this instruction has a single operand. It computes the following result:

dest = NOT(*dest*)

The *dest* operand (for *not*) must be a register or memory operand. This instruction inverts all the bits in the specified destination operand.

The following program inputs two hexadecimal values from the user and calculates their logical AND, OR, XOR, and NOT:

```
program LogicalOp;
#include( "stdlib.hhf" );
begin LogicalOp;

    stdout.put( "Input left operand: " );
    stdin.get( eax );
    stdout.put( "Input right operand: " );
    stdin.get( ebx );

    mov( eax, ecx );
    and( ebx, ecx );
    stdout.put( "$", eax, " AND $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    or( ebx, ecx );
    stdout.put( "$", eax, " OR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    xor( ebx, ecx );
    stdout.put( "$", eax, " XOR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    not( ecx );
    stdout.put( "NOT $", eax, " = $", ecx, nl );

    mov( ebx, ecx );
    not( ecx );
    stdout.put( "NOT $", ebx, " = $", ecx, nl );

end LogicalOp;
```

Program 3.15 AND, OR, XOR, and NOT Example

3.9 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. 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 values. Negative values are objects in their own right, just like positive numbers; therefore, we'll have to use some of the 256 different eight-bit values to represent negative numbers. In other words, we've got to use up some of the (unsigned) 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 and zero. So we can represent the negative values -128..-1 and the non-negative values 0..127 with a single eight bit byte. 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:

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

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

$7FFF is positive.

$FFFF is negative.

$FFF 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.

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, %0000_0101, 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:

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

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

$4000: +16,384.

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

```
$7FFF:      %0111_1111_1111_1111   +32,767
            %1000_0000_0000_0000   Invert all the bits (8000h)
            %1000_0000_0000_0001   Add one (8001h or -32,767)

4000h:      %0100_0000_0000_0000   16,384
            %1011_1111_1111_1111   Invert all the bits ($BFFF)
            %1100_0000_0000_0000   Add one ($C000 or -16,384)

$8000:      %1000_0000_0000_0000   -32,768
            %0111_1111_1111_1111   Invert all the bits ($7FFF)
            %1000_0000_0000_0000   Add one (8000h or -32768)
```

$8000 inverted becomes $7FFF. After adding one we obtain $8000! 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.

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:

```
%  0000_0101
%  1111_1011
------------
%1_0000_0000
```

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 associated with this chapter, you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction, NEG (negate), that 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 %1100_0000 could represent an IBM/ASCII character, it could represent the unsigned decimal value 192, or it could represent the signed decimal value -64. As the programmer, it is your responsibility to use this data consistently.

The 80x86 negate instruction, NEG, uses the same syntax as the NOT instruction; that is, it takes a single destination operand:

```
neg( dest );
```

This instruction computes "dest = -dest;" and the operand has the same limitations as for NOT (it must be a memory location or a register). NEG operates on byte, word, and dword-sized objects. Of course, since this is a signed integer operation, it only makes sense to operate on signed integer values. The following program demonstrates the two's complement operation by using the NEG instruction:

```
program twosComplement;
#include( "stdlib.hhf" );

static
```



```
    PosValue:   int8;
    NegValue:   int8;

begin twosComplement;

    stdout.put( "Enter an integer between 0 and 127: " );
    stdin.get( PosValue );

    stdout.put( nl, "Value in hexadecimal: $" );
    stdout.putb( PosValue );

    mov( PosValue, al );
    not( al );
    stdout.put( nl, "Invert all the bits: $", al, nl );
    add( 1, al );
    stdout.put( "Add one: $", al, nl );
    mov( al, NegValue );
    stdout.put( "Result in decimal: ", NegValue, nl );

    stdout.put
    (
        nl,
        "Now do the same thing with the NEG instruction: ",
        nl
    );
    mov( PosValue, al );
    neg( al );
    mov( al, NegValue );
    stdout.put( "Hex result = $", al, nl );
    stdout.put( "Decimal result = ", NegValue, nl );

end twosComplement;
```

Program 3.16 The Two's Complement Operation

As you saw in the previous chapters, you use the *int8*, *int16*, and *int32* data types to reserve storage for signed integer variables. Those chapters also introduced routines like *stdout.puti8* and *stdin.geti32* that read and write signed integer values. Since this section has made it abundantly clear that you must differentiate signed and unsigned calculations in your programs, you should probably be asking yourself about now "how do I declare and use unsigned integer variables?"

The first part of the question, "how do you declare unsigned integer variables," is the easiest to answer. You simply use the *uns8*, *uns16*, and *uns32* data types when declaring the variables, for example:

```
static
    u8:     uns8;
    u16:    uns16;
    u32:    uns32;
```

As for using these unsigned variables, the HLA Standard Library provides a complementary set of input/output routines for reading and displaying unsigned variables. As you can probably guess, these routines include *stdout.putu8*, *stdout.putu16*, *stdout.putu32*, *stdout.putu8Size*, *stdout.putu16Size*, *stdout.putu32Size*, *stdin.getu8*, *stdin.getu16*, and *stdin.getu32*. You use these routines just as you would use their signed integer counterparts except, of course, you get to use the full range of the unsigned values with these routines. The following source code demonstrates unsigned I/O as well as demonstrating what can happen if you mix signed and unsigned operations in the same calculation:


```
program UnsExample;
#include( "stdlib.hhf" );

static
    UnsValue:   uns16;

begin UnsExample;

    stdout.put( "Enter an integer between 32,768 and 65,535: " );
    stdin.getu16();
    mov( ax, UnsValue );

    stdout.put
    (
        "You entered ",
        UnsValue,
        ".  If you treat this as a signed integer, it is "
    );
    stdout.puti16( UnsValue );
    stdout.newln();

end UnsExample;
```

Program 3.17 Unsigned I/O

3.10 Sign Extension, Zero Extension, Contraction, and Saturation

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 $C0. The 16-bit equivalent of this number is $FFC0. Now consider the value "+64". The eight and 16 bit versions of this value are $40 and $0040, respectively. 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 $FF; 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.

You must use sign extension 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.

```
Sign Extension:
Eight Bits  Sixteen Bits  Thirty-two Bits

 $80        $FF80         $FFFF_FF80
 $28        $0028         $0000_0028
 $9A        $FF9A         $FFFF_FF9A
```



```
$7F         $007F          $0000_007F
---         $1020          $0000_1020
---         $8086          $FFFF_8086
```

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 larger operand. For example, to zero extend the value $82 to 16-bits you simply add a zero to the H.O. byte yielding $0082.

```
Zero Extension:
Eight Bits    Sixteen Bits           Thirty-two Bits

 $80            $0080                 $0000_0080
 $28            $0028                 $0000_0028
 $9A            $009A                 $0000_009A
 $7F            $007F                 $0000_007F
 ---            $1020                 $0000_1020
 ---            $8086                 $0000_8086
```

The 80x86 provides several instructions that will let you sign or zero extend a smaller number to a larger number. The first group of instructions we will look at will sign extend the AL, AX, or EAX register. These instructions are

- cbw(); // Converts the byte in AL to a word in AX via sign extension.
- cwd(); // Converts the word in AX to a double word in DX:AX
- cdq(); // Converts the double word in EAX to the quad word in EDX:EAX
- cwde(); // Converts the word in AX to a doubleword in EAX.

Note that the CWD (convert word to doubleword) instruction does not sign extend the word in AX to the doubleword in EAX. Instead, it stores the H.O. doubleword of the sign extension into the DX register (the notation “DX:AX” tells you that you have a double word value with DX containing the upper 16 bits and AX containing the lower 16 bits of the value). If you want the sign extension of AX to go into EAX, you should use the CWDE (convert word to doubleword, extended) instruction.

The four instructions above are unusual in the sense that these are the first instructions you’ve seen that do not have any operands. These instructions’ operands are *implied* by the instructions themselves.

Within a few chapters you will discover just how important these instructions are, and why the CWD and CDQ instructions involve the DX and EDX registers. However, for simple sign extension operations, these instructions have a few major drawbacks - you do not get to specify the source and destination operands and the operands must be registers.

For general sign extension operations, the 80x86 provides an extension of the MOV instruction, MOVSX (move with sign extension), that copies data and sign extends the data while copying it. The MOVSX instruction’s syntax is very similar to the MOV instruction:

```
movsx( source, dest );
```

The big difference in syntax between this instruction and the MOV instruction is the fact that the destination operand must be larger than the source operand. That is, if the source operand is a byte, the destination operand must be a word or a double word. Likewise, if the source operand is a word, the destination operand must be a double word. Another difference is that the destination operand has to be a register; the source operand, however, can be a memory location[4].

To zero extend a value, you can use the MOVZX instruction. It has the same syntax and restrictions as the MOVSX instruction. Zero extending certain eight-bit registers (AL, BL, CL, and DL) into their corresponding 16-bit registers is easily accomplished without using MOVZX by loading the complementary H.O.

4. This doesn’t turn out to be much of a limitation because sign extension almost always precedes an arithmetic operation which must take place in a register.

register (AH, BH, CH, or DH) with zero. Obviously, to zero extend AX into DX:AX or EAX into EDX:EAX, all you need to do is load DX or EDX with zero[5].

The following sample program demonstrates the use of the sign extension instructions:

```
program signExtension;
#include( "stdlib.hhf" );

static
    i8:     int8;
    i16:    int16;
    i32:    int32;

begin signExtension;

    stdout.put( "Enter a small negative number: " );
    stdin.get( i8 );

    stdout.put( nl, "Sign extension using CBW and CWDE:", nl, nl );

    mov( i8, al );
    stdout.put( "You entered ", i8, " ($", al, ")", nl );

    cbw();
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    cwde();
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

    stdout.put( nl, "Sign extension using MOVSX:", nl, nl );

    movsx( i8, ax );
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    movsx( i8, eax );
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

end signExtension;
```

Program 3.18 Sign Extension Instructions

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 $FE40. Unfortunately, the magnitude of this number is too large 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.

5. Zero extending into DX:AX or EDX:EAX is just as necessary as the CWD and CDQ instructions, as you will eventually see.

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 $FF. 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):

```
$FF80 can be sign contracted to $80.
$0040 can be sign contracted to $40.
$FE40 cannot be sign contracted to 8 bits.
$0100 cannot be sign contracted to 8 bits.
```

Another way to reduce the size of an integer is through saturation. Saturation is useful in situations where you must convert a larger object to a smaller object and you're willing to live with possible loss of precision. To convert a value via saturation you simply copy the larger value to the smaller value if it is not outside the range of the smaller object. If the larger value is outside the range of the smaller value, then you *clip* the value by setting it to the largest (or smallest) value within the range of the smaller object.

For example, when converting a 16-bit signed integer to an eight-bit signed integer, if the 16-bit value is in the range -128..+127 you simply copy the L.O. byte of the 16-bit object to the eight-bit object. If the 16-bit signed value is greater than +127, then you clip the value to +127 and store +127 into the eight-bit object. Likewise, if the value is less than -128, you clip the final eight bit object to -128. Saturation works the same way when clipping 32-bit values to smaller values. If the larger value is outside the range of the smaller value, then you simply set the smaller value to the value closest to the out of range value that you can represent with the smaller value.

Obviously, if the larger value is outside the range of the smaller value, then there will be a loss of precision during the conversion. While clipping the value to the limits the smaller object imposes is never desirable, sometimes this is acceptable as the alternative is to raise an exception or otherwise reject the calculation. For many applications, such as audio or video processing, the clipped result is still recognizable, so this is a reasonable conversion to use.

3.11 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 3.8).

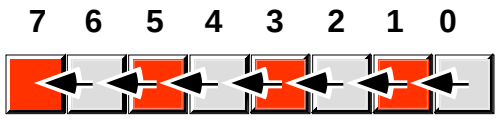

Figure 3.8 Shift Left Operation

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?" We'll shift a zero into bit zero and the previous value of bit seven will be the *carry* out of this operation.

The 80x86 provides a shift left instruction, SHL, that performs this useful operation. The syntax for the SHL instruction is the following:


```
shl( count, dest );
```

The count operand is either “CL” or a constant in the range 0..n, where n is one less than the number of bits in the destination operand (i.e., n=7 for eight-bit operands, n=15 for 16-bit operands, and n=31 for 32-bit operands). The dest operand is a typical dest operand, it can be either a memory location or a register.

When the count operand is the constant one, the SHL instruction does the following:

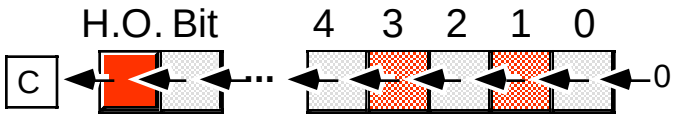

Figure 3.9 Operation of the SHL(1, Dest) Instruction

In Figure 3.9, the “C” represents the carry flag. That is, the bit shifted out of the H.O. bit of the operand is moved into the carry flag. Therefore, you can test for overflow after a SHL(1, dest) instruction by testing the carry flag immediately after executing the instruction (e.g., by using “if(@c) then...” or “if(@nc) then...”).

Intel’s literature suggests that the state of the carry flag is undefined if the shift count is a value other than one. Usually, the carry flag contains the last bit shifted out of the destination operand, but Intel doesn’t seem to guarantee this. If you need to shift more than one bit out of an operand and you need to capture all the bits you shift out, you should take a look at the SHLD and SHRD instructions in the appendicies.

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 means shift one digit position to the left)
```

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 3.10).

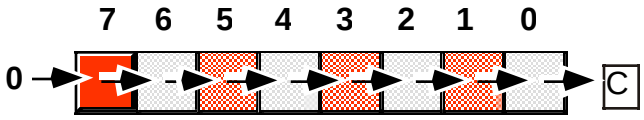

Figure 3.10 Shift Right Operation

As you would probably expect by now, the 80x86 provides a SHR instruction that will shift the bits to the right in a destination operand. The syntax is the same as the SHL instruction except, of course, you specify SHR rather than SHL:

```
SHR( count, dest );
```


This instruction shifts a zero into the H.O. bit of the destination operand, it shifts all the other bits one place to the right (that is, from a higher bit number to a lower bit number). Finally, bit zero is shifted into the carry flag. If you specify a count of one, the SHR instruction does the following:

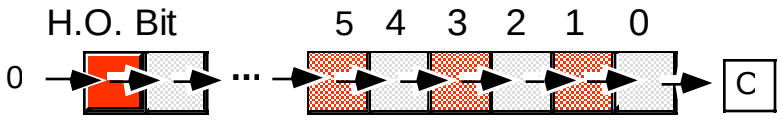

Figure 3.11 SHR(1, Dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

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.

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*[6]. 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 3.12 shows.

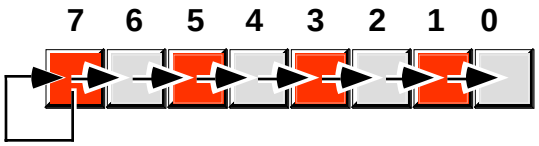

Figure 3.12 Arithmetic Shift Right Operation

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, it's just uses a diffferent (though valid) definition of integer division.

6. 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.

The 80x86 provides an arithmetic shift right instruction, SAR (shift arithmetic right). This instruction's syntax is nearly identical to SHL and SHR. The syntax is

```
SAR( count, dest );
```

The usual limitations on the count and destination operands apply. This instruction does the following if the count is one:

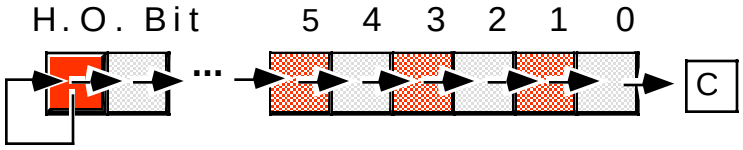

Figure 3.13 SAR(1, dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

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.

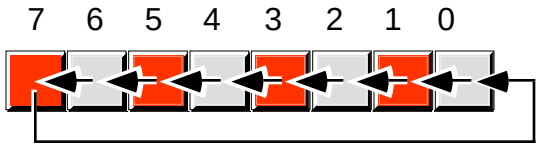

Figure 3.14 Rotate Left Operation

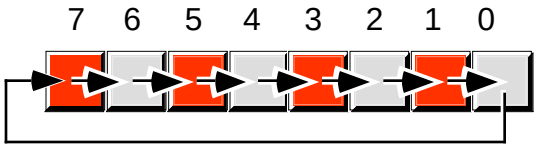

Figure 3.15 Rotate Right Operation

The 80x86 provides ROL (rotate left) and ROR (rotate right) instructions that do these basic operations on their operands. The syntax for these two instructions is similar to the shift instructions:

```
rol( count, dest );
ror( count, dest );
```


Once again, this instructions provide a special behavior if the shift count is one. Under this condition these two instructions also copy the bit shifted out of the destination operand into the carry flag as the following two figures show:

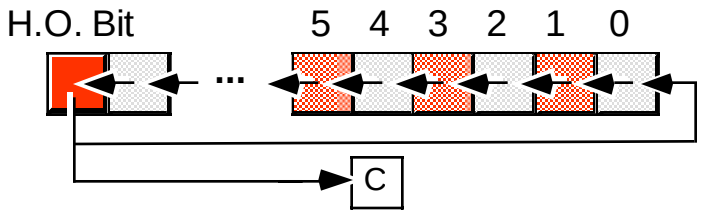

Figure 3.16 ROL(1, Dest) Operation

Note that, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

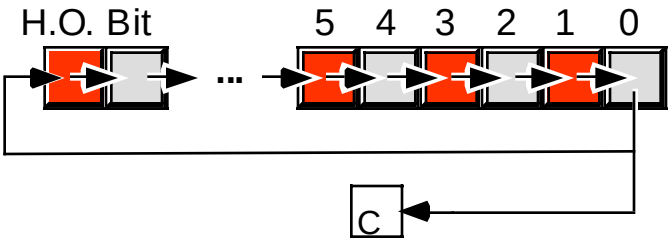

Figure 3.17 ROR(1, Dest) Operation

It will turn out that it is often more convenient for the rotate operation to shift the output bit through the carry and shift the previous carry value back into the input bit of the shift operation. The 80x86 RCL (rotate through carry left) and RCR (rotate through carry right) instructions achieve this for you. These instructions use the following syntax:

```
RCL( count, dest );
RCR( count, dest );
```

As is true for the other shift and rotate instructions, the count operand is either a constant or the CL register and the destination operand is a memory location or register. The count operand must be a value that is less than the number of bits in the destination operand. For a count value of one, these two instructions do the following:

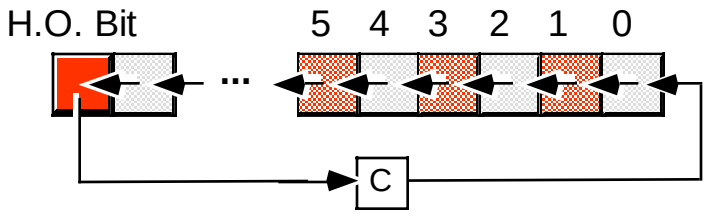

Figure 3.18 RCL(1, Dest) Operation

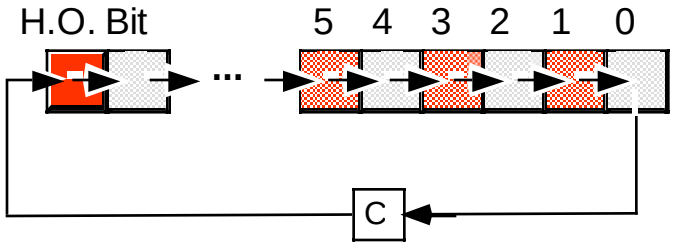

Figure 3.19 RCR(1, Dest) Operation

Again, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

3.12 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 "04/02/01". 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 the following figure:

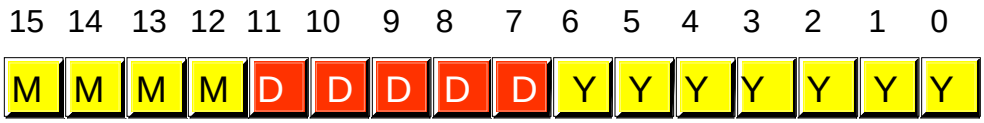

Figure 3.20 Short Packed Date Format (Two Bytes)

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, 2001 would be represented as $4101:

```
0100    00010 0000001 = %0100_0001_0000_0001 or $4101
 4      2       01
```

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. The following sample program demonstrates the effort that must go into packing and unpacking this 16-bit date format:

```
program dateDemo;

#include( "stdlib.hhf" );

static
    day:        uns8;
    month:      uns8;
    year:       uns8;

    packedDate: word;

begin dateDemo;

    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    //  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    //   m  m  m  m  d  d  d  d  d  y  y  y  y  y  y  y

    mov( 0, ax );
    mov( ax, packedDate );  //Just in case there is an error.
    if( month > 12 ) then

        stdout.put( "Month value is too large", nl );

    elseif( month = 0 ) then

        stdout.put( "Month value must be in the range 1..12", nl );

    elseif( day > 31 ) then

        stdout.put( "Day value is too large", nl );
```



```
    elseif( day = 0 ) then

        stdout.put( "Day value must be in the range 1..31", nl );

    elseif( year > 99 ) then

        stdout.put( "Year value must be in the range 0..99", nl );

    else

        mov( month, al );
        shl( 5, ax );
        or( day, al );
        shl( 7, ax );
        or( year, al );
        mov( ax, packedDate );

    endif;

    // Okay, display the packed value:

    stdout.put( "Packed data = $", packedDate, nl );



    // Unpack the date:

    mov( packedDate, ax );
    and( $7f, al );         // Retrieve the year value.
    mov( al, year );

    mov( packedDate, ax );  // Retrieve the day value.
    shr( 7, ax );
    and( %1_1111, al );
    mov( al, day );

    mov( packedDate, ax );  // Retrive the month value.
    rol( 4, ax );
    and( %1111, al );
    mov( al, month );

    stdout.put( "The date is ", month, "/", day, "/", year, nl );



end dateDemo;
```

Program 3.19 Packing and Unpacking Date Data

Of course, having gone through the problems with Y2K, using a date format that limits you to 100 years (or even 127 years) would be quite foolish at this time. If you're concerned about your software running 100 years from now, perhaps it would be wise to use a three-byte date format rather than a two-byte format. As you will see in the chapter on arrays, however, you should always try to create data objects whose length is an even power of two (one byte, two bytes, four bytes, eight bytes, etc.) or you will pay a performance penalty. Hence, it is probably wise to go ahead and use four bytes and pack this data into a dword variable. Figure 3.21 shows a possible data organization for a four-byte date.

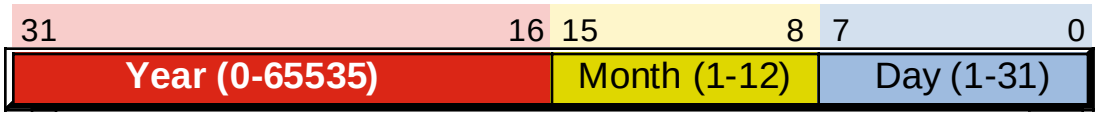

Figure 3.21 Long Packed Date Format (Four Bytes)

In this long packed data format several changes were made beyond simply extending the number of bits associated with the year. First, since there are lots of extra bits in a 32-bit dword variable, this format allots extra bits to the month and day fields. Since these two fields consist of eight bits each, they can be easily extracted as a byte object from the dword. This leaves fewer bits for the year, but 65,536 years is probably sufficient; you can probably assume without too much concern that your software will not still be in use 63 thousand years from now when this date format will wrap around.

Of course, you could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into one byte each and one that should probably have at least two bytes. Since this "packed" date format consumes the same four bytes as the unpacked version, what is so special about this format? Well, another difference you will note between this long packed date format and the short date format appearing in Figure 3.20 is the fact that this long date format rearranges the bits so the *Year* is in the H.O. bit positions, the *Month* field is in the middle bit positions, and the *Day* field is in the L.O. bit positions. This is important because it allows you to very easily compare two dates to see if one date is less than, equal to, or greater than another date. Consider the following code:

```
mov( Date1, eax );       // Assume Date1 and Date2 are dword variables
if( eax > Date2 ) then   //  using the Long Packed Date format.

    << do something if Date1 > Date2 >>

endif;
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare *Date1* and *Date2* in such a straight-forward fashion. Therefore, this example demonstrates another reason for packing data even if you don't realize any space savings- it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

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. Of course, a classic example of packed data is the FLAGs register (see Figure 3.22). This register packs nine important boolean objects (along with seven important system flags) into a single 16-bit register. You will commonly need to access many of these flags. For this reason, the 80x86 instruction set provides many ways to manipulate the individual bits in the FLAGs register. Of course, you can test many of the condition code flags using the HLA @c, @nc, @z, @nz, etc., pseudo-boolean variables in an IF statement or other statement using a boolean expression.

In addition to the condition codes, the 80x86 provides instructions that directly affect certain flags. These instructions include the following:

- cld(); Clears (sets to zero) the direction flag.
- std(); Sets (to one) the direction flag.
- cli(); Clears the interrupt disable flag.
- sti(); Sets the interrupt disable flag.
- clc(); Clears the carry flag.
- stc(); Sets the carry flag.
- cmc(); Complements (inverts) the carry flag.
- sahf(); Stores the AH register into the L.O. eight bits of the FLAGs register.
- lahf(); Loads AH from the L.O. eight bits of the FLAGs register.

There are other instructions that affect the FLAGs register as well; these, however, demonstrate how to access several of the packed boolean values in the FLAGs register. The LAHF and SAHF instructions, in particular, provide a convenient way to access the L.O. eight bits of the FLAGs register as an eight-bit byte (rather than as eight separate one-bit values).

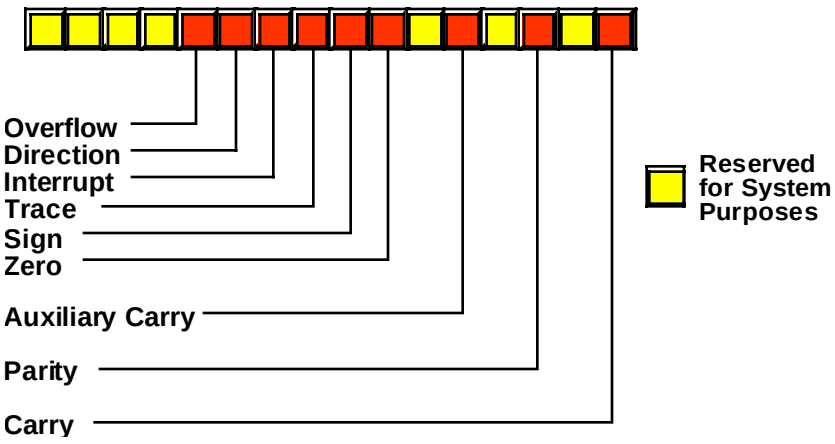

Figure 3.22 The FLAGs Register as a Packed Data Type

The LAHF (load AH with the L.O. eight bits of the FLAGs register) and the SAHF (store AH into the L.O. byte of the FLAGs register) use the following syntax:

```
lahf();
sahf();
```

3.13 Putting It All Together

In this chapter you've seen how we represent numeric values inside the computer. You've seen how to represent values using the decimal, binary, and hexadecimal numbering systems as well as the difference between signed and unsigned numeric representation. Since we represent nearly everything else inside a computer using numeric values, the material in this chapter is very important. Along with the base representation of numeric values, this chapter discusses the finite bit-string organization of data on typical computer systems, specfically bytes, words, and doublewords. Next, this chapter discusses arithmetic and logical operations on the numbers and presents some new 80x86 instructions to apply these operations to values inside the CPU. Finally, this chapter concludes by showing how you can pack several different numeric values into a fixed-length object (like a byte, word, or doubleword).

Absent from this chapter is any discussion of non-integer data. For example, how do we represent real numbers as well as integers? How do we represent characters, strings, and other non-numeric data? Well, that's the subject of the next chapter, so keep on reading...

