Bit Manipulation Chapter Five

5.1 Chapter Overview

Manipulating bits in memory is, perhaps, the thing that assembly language is most famous for. Indeed, one of the reasons people claim that the “C” programming language is a “medium-level” language rather than a high level language is because of the vast array of bit manipulation operators that it provides. Even with this wide array of bit manipulation operations, the C programming language doesn’t provide as complete a set of bit manipulation operations as assembly language.

This chapter will discuss how to manipulate strings of bits in memory and registers using 80x86 assembly language. This chapter begins with a review of the bit manipulation instructions covered thus far and it also introduces a few new instructions. This chapter reviews information on packing and unpacking bit strings in memory since this is the basis for many bit manipulation operations. Finally, this chapter discusses several bit-centric algorithms and their implementation in assembly language.

5.2 What is Bit Data, Anyway?

Before describing how to manipulate bits, it might not be a bad idea to define exactly what this text means by “bit data.” Most readers probably assume that “bit manipulation programs” twiddle individual bits in memory. While programs that do this are definitely “bit manipulation programs,” we’re not going to limit this title to just those programs. For our purposes, bit manipulation refers to working with data types that consist of strings of bits that are non-contiguous or are not an even multiple of eight bits long. Generally, such bit objects will not represent numeric integers, although we will not place this restriction on our bit strings.

A *bit string* is some contiguous sequence of one or more bits (this term even applies if the bit string’s length is an even multiple of eight bits). Note that a bit string does not have to start or end at any special point. For example, a bit string could start in bit seven of one byte in memory and continue through to bit six of the next byte in memory. Likewise, a bit string could begin in bit 30 of EAX, consume the upper two bits of EAX, and then continue from bit zero through bit 17 of EBX. In memory, the bits must be physically contiguous (i.e., the bit numbers are always increasing except when crossing a byte boundary, and at byte boundaries the byte number increases by one). In registers, if a bit string crosses a register boundary, the application defines the continuation register but the bit string always continues in bit zero of that second register.

A *bit set* is a collection of bits, not necessarily contiguous (though it may be), within some larger data structure. For example, bits 0..3, 7, 12, 24, and 31 from some double word object forms a set of bits. Usually, we will limit bit sets to some reasonably sized *container object* (that is, the data structure that encapsulates the bit set), but the definition doesn’t specifically limit the size. Normally, we will deal with bit sets that are part of an object no more than about 32 or 64 bits in size. Note that bit strings are special cases of bit sets.

A *bit run* is a sequence of bits with all the same value. A *run of zeros* is a bit string containing all zeros, a *run of ones* is a bit string containing all ones. The *first set bit* in a bit string is the bit position of the first bit containing a one in a bit string, i.e., the first ‘1’ bit following a possible run of zeros. A similar definition exists for the *first clear bit*. The *last set bit* is the last bit position in a bit string containing that contains ‘1’; afterwards, the remainder of the string forms an uninterrupted run of zeros. A similar definition exists for the *last clear bit*.

A *bit offset* is the number of bits from some boundary position (usually a byte boundary) to the specified bit. As noted in Volume One, we number the bits starting from zero at the boundary location. If the offset is less than 32, then the bit offset is the same as the bit number in a byte, word, or double word value.

A *mask* is a sequence of bits that we'll use to manipulate certain bits in another value. For example, the bit string %0000_1111_0000, when used with the AND instruction, can mask away (clear) all the bits except bits four through seven. Likewise, if you use this same value with the OR instruction, it can force bits four through seven to ones in the destination operand. The term "mask" comes from the use of these bit strings with the AND instruction; in those situations the one and zero bits behave like masking tape when you're painting something; they pass through certain bits unchanged while masking out the other bits.

Armed with these definitions, we're ready to start manipulating some bits!

5.3 Instructions That Manipulate Bits

Bit manipulation generally consists of six activities: setting bits, clearing bits, inverting bits, testing and comparing bits, extracting bits from a bit string, and inserting bits into a bit string. By now you should be familiar with most of the instructions we'll use to perform these operations; their introduction started way back in the earliest chapters of Volume One. Nevertheless, it's worthwhile to review the old instructions here as well as present the few bit manipulation instructions we've yet to consider.

The most basic bit manipulation instructions are the AND, OR, XOR, NOT, TEST, and shift and rotate instructions. Indeed, on the earliest 80x86 processors, these were the only instructions available for bit manipulation. The following paragraphs review these instructions, concentrating on how you could use them to manipulate bits in memory or registers.

The AND instruction provides the ability to strip away unwanted bits from some bit sequence, replacing the unwanted bits with zeros. This instruction is especially useful for isolating a bit string or a bit set that is merged with other, unrelated data (or, at least, data that is not part of the bit string or bit set). For example, suppose that a bit string consumes bit positions 12 through 24 of the EAX register, we can isolate this bit string by setting all other bits in EAX to zero by using the following instruction:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
```

Most programs use the AND instruction to clear bits that are not part of the desired bit string. In theory, you could use the OR instruction to mask all unwanted bits to ones rather than zeros, but later comparisons and operations are often easier if the unneeded bit positions contain zero.

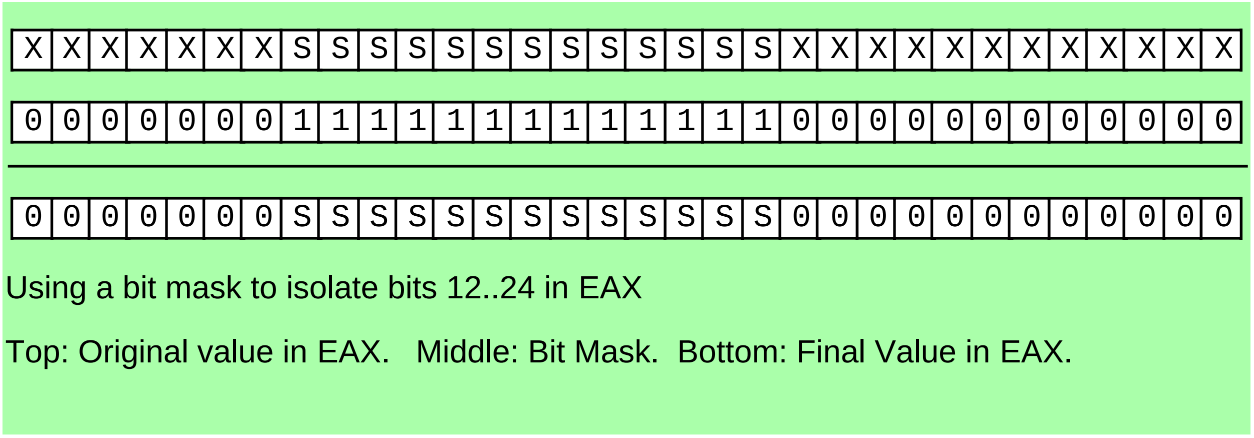

X X X X X X X S S S S S S S S S S S S S X X X X X X X X X X X X
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 S S S S S S S S S S S S S 0 0 0 0 0 0 0 0 0 0 0 0
Using a bit mask to isolate bits 12..24 in EAX
Top: Original value in EAX. Middle: Bit Mask. Bottom: Final Value in EAX.

Figure 5.1 Isolating a Bit String Using the AND Instruction

Once you've cleared the unneeded bits in a set of bits, you can often operate on the bit set in place. For example, to see if the string of bits in positions 12 through 24 of EAX contain $12F3 you could use the following code:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, %1_0010_1111_0011_0000_0000_0000 );
```

Here's another solution, using constant expressions, that's a little easier to digest:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, $12F3 << 12 );  // "<<12" shifts $12F3 to the left 12 bits.
```

Most of the time, however, you'll want (or need) the bit string aligned with bit zero in EAX prior to any operations you would want to perform. Of course, you can use the SHR instruction to properly align the value after you've masked it:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
shr( 12, eax );
cmp( eax, $12F3 );
<< Other operations that requires the bit string at bit #0 >>
```

Now that the bit string is aligned to bit zero, the constants and other values you use in conjunction with this value are easier to deal with.

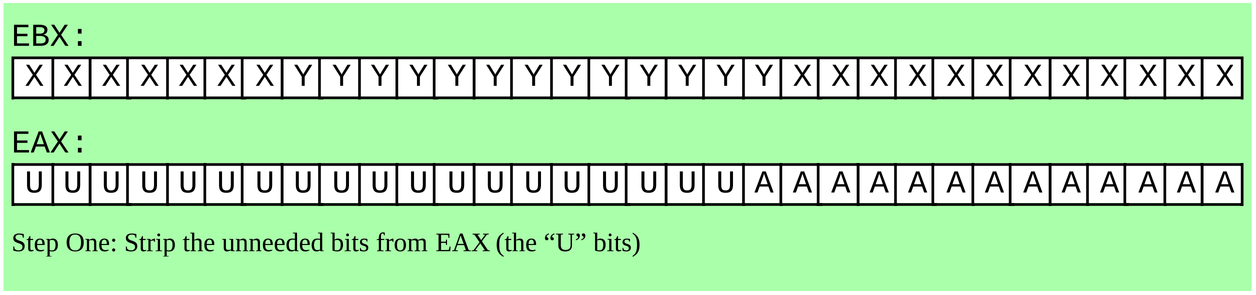
You can also use the OR instruction to mask unwanted bits around a set of bits. However, the OR instruction does not let you clear bits, it allows you to set bits to ones. In some instances setting all the bits around your bit set may be desirable; most software, however, is easier to write if you clear the surrounding bits rather than set them.

The OR instruction is especially useful for inserting a bit set into some other bit string. To do this, there are several steps you must go through:

- Clear all the bits surrounding your bit set in the source operand.
- Clear all the bits in the destination operand where you wish to insert the bit set.
- OR the bit set and destination operand together.

For example, suppose you have a value in bits 0..12 of EAX that you wish to insert into bits 12..24 of EBX without affecting any of the other bits in EBX. You would begin by stripping out bits 13 and above from EAX; then you would strip out bits 12..24 in EBX. Next, you would shift the bits in EAX so the bit string occupies bits 12..24 of EAX. Finally, you would OR the value in EAX into EBX (see Figure 5.2):

```
and( $1FFF, eax );      // Strip all but bits 0..12 from EAX
and( $1FF_F000, ebx );  // Clear bits 12..24 in EBX.
shl( 12, eax );         // Move bits 0..12 to 12..24 in EAX.
or( eax, ebx );         // Merge the bits into EBX.
```


EBX:
X X X X X X X Y Y Y Y Y Y Y Y Y Y Y Y Y X X X X X X X X X X X X
EAX:
U U U U U U U U U U U U U U U U U U U A A A A A A A A A A A A A
Step One: Strip the unneeded bits from EAX (the "U" bits)

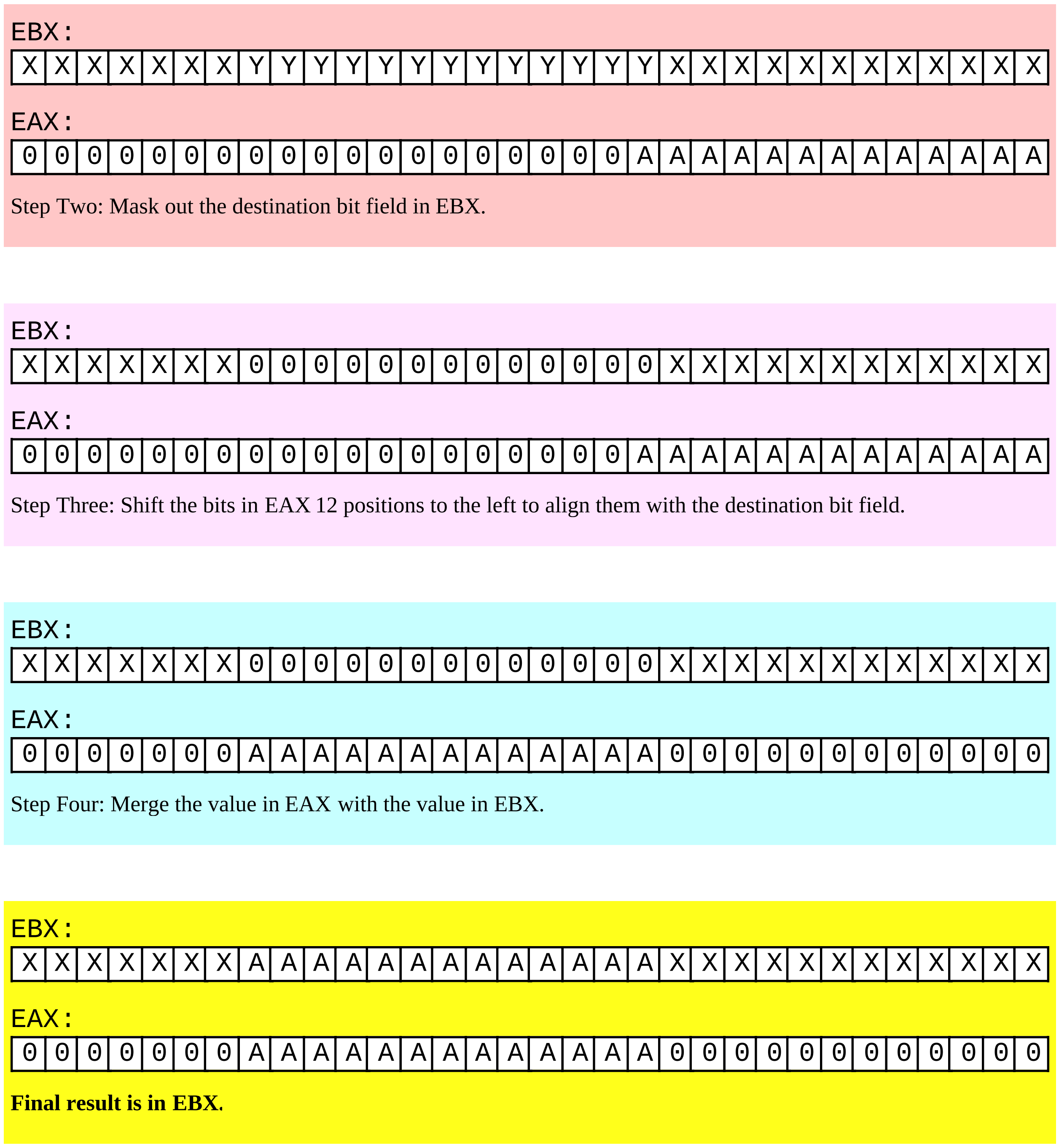

EBX:
X X X X X X X Y Y Y Y Y Y Y Y Y Y Y Y Y X X X X X X X X X X X X
EAX:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 A A A A A A A A A A A A A
Step Two: Mask out the destination bit field in EBX.
EBX:
X X X X X X X 0 0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X X X X X
EAX:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 A A A A A A A A A A A A A
Step Three: Shift the bits in EAX 12 positions to the left to align them with the destination bit field.
EBX:
X X X X X X X 0 0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X X X X X
EAX:
0 0 0 0 0 0 0 A A A A A A A A A A A A A 0 0 0 0 0 0 0 0 0 0 0 0
Step Four: Merge the value in EAX with the value in EBX.
EBX:
X X X X X X X A A A A A A A A A A A A A X X X X X X X X X X X X
EAX:
0 0 0 0 0 0 0 A A A A A A A A A A A A A 0 0 0 0 0 0 0 0 0 0 0 0
Final result is in EBX.

Figure 5.2 Inserting Bits 0..12 of EAX into Bits 12..24 of EBX

In this example the desired bits (AAAAAAAAAAAAA) formed a bit string. However, this algorithm still works fine even if you're manipulating a non-contiguous set of bits. All you've got to do is to create an appropriate bit mask you can use for ANDing that has ones in the appropriate places.

When working with bit masks, it is incredibly poor programming style to use literal numeric constants as in the past few examples. You should always create symbolic constants in the HLA CONST (or VAL) section for your bit masks. Combined with some constant expressions, you can produce code that is much easier to read and maintain. The current example code is more properly written as:

```
const
    StartPosn: 12;
    BitMask: dword := $1FFF << StartPosn;   // Mask occupies bits 12..24
        .
        .
        .
        shl( StartPosn, eax );   // Move into position.
        and( BitMask, eax );     // Strip all but bits 12..24 from EAX
        and( !BitMask, ebx ); // Clear bits 12..24 in EBX.
        or( eax, ebx );          // Merge the bits into EBX.
```

Notice the use of the compile-time not operator ("!") to invert the bit mask in order to clear the bit positions in EBX where the code inserts the bits from EAX. This saves having to create another constant in the program that has to be changed anytime you modify the *BitMask* constant. Having to maintain two separate symbols whose values are dependent on one another is not a good thing in a program.

Of course, in addition to merging one bit set with another, the OR instruction is also useful for forcing bits to one in a bit string. By setting various bits in a source operand to one you can force the corresponding bits in the destination operand to one by using the OR instruction.

The XOR instruction, as you may recall, gives you the ability to invert selected bits belonging to a bit set. Although the need to invert bits isn't as common as the need to set or clear them, the XOR instruction still sees considerable use in bit manipulation programs. Of course, if you want to invert all the bits in some destination operand, the NOT instruction is probably more appropriate than the XOR instruction; however, to invert selected bits while not affecting others, the XOR is the way to go.

One interesting fact about XOR's operation is that it lets you manipulate known data in just about any way imaginable. For example, if you know that a field contains %1010 you can force that field to zero by XORing it with %1010. Similarly, you can force it to %1111 by XORing it with %0101. Although this might seem like a waste, since you can easily force this four-bit string to zero or all ones using AND or OR, the XOR instruction has two advantages: (1) you are not limited to forcing the field to all zeros or all ones;, you can actually set these bits to any of the 16 valid combinations via XOR; (2) if you need to manipulate other bits in the destination operand at the same time, AND/OR may not be able to accommodate you. For example, suppose that you know that one field contains %1010 that you want to force to zero and another field contains %1000 and you wish to increment that field by one (i.e., set the field to %1001). You cannot accomplish both operations with a single AND or OR instruction, but you can do this with a single XOR instruction; just XOR the first field with %1010 and the second field with %0001. Remember, however, that this trick only works if you know the current value of a bit set within the destination operand. Of course, while you're adjusting the values of bit fields containing known values, you can invert bits in other fields simultaneously.

In addition to setting, clearing, and inverting bits in some destination operand, the AND, OR, and XOR instructions also affect various condition codes in the FLAGs register. These instructions affect the flags as follows:

- These instructions always clear the carry and overflow flags.
- These instructions set the sign flag if the result has a one in the H.O. bit; they clear it otherwise. I.e., these instructions copy the H.O. bit of the result into the sign flag.
- These instructions set/clear the zero flag depending on whether the result is zero.
- These instructions set the parity flag if there are an even number of set bits in the L.O. byte of the destination operand, they clear the parity flag if there are an odd number of one bits in the L.O. byte of the destination operand.

The first thing to note is that these instructions always clear the carry and overflow flags. This means that you cannot expect the system to preserve the state of these two flags across the execution of these instructions. A very common mistake in many assembly language programs is the assumption that these instructions do not affect the carry flag. Many people will execute an instruction that sets/clears the carry flag, execute an AND/OR/XOR instruction, and then attempt to test the state of the carry from the previous instruction. This simply will not work.

One of the more interesting aspects to these instructions is that they copy the H.O. bit of their result into the sign flag. This means that you can easily test the setting of the H.O. bit of the result by testing the sign flag (using SETS/SETNS, JS/JNS, or by using the @S/@NS flags in a boolean expression). For this reason, many assembly language programmers will often place an important boolean variable in the H.O. bit of some operand so they can easily test the state of that bit using the sign flag after a logical operation.

We haven't talked much about the parity flag in this text. Indeed, earlier volumes have done little more than acknowledge its existence. We're not going to get into a big discussion of this flag and what you use it for since the primary purpose for this flag has been taken over by hardware[1]. However, since this is a chapter on bit manipulation and parity computation is a bit manipulation operation, it seems only fitting to provide a brief discussion of the parity flag at this time.

Parity is a very simple error detection scheme originally employed by telegraphs and other serial communication schemes. The idea was to count the number of set bits in a character and include an extra bit in the transmission to indicate whether that character contained an even or odd number of set bits. The receiving end of the transmission would also count the bits and verify that the extra "parity" bit indicated a successful transmission. We're not going to explore the information theory aspects of this error checking scheme at this point other than to point out that the purpose of the parity flag is to help compute the value of this extra bit.

The 80x86 AND, OR, and XOR instructions set the parity bit if the L.O. byte of their operand contains an even number of set bits. An important fact bears repeating here: the parity flag only reflects the number of set bits in the L.O. byte of the destination operand; it does not include the H.O. bytes in a word, double word, or other sized operand. The instruction set only uses the L.O. byte to compute the parity because communication programs that use parity are typically character-oriented transmission systems (there are better error checking schemes if you transmit more than eight bits at a time).

Although the need to know whether the L.O. (or only) byte of some computation has an even or odd number of set bits isn't common in modern programs, it does come in useful once in a great while. Since this is, intrinsically, a bit operation, it's worthwhile to mention the use of this flag and how the AND/OR/XOR instructions affect this flag.

The zero flag setting is one of the more important results the AND/OR/XOR instructions produce. Indeed, programs reference this flag so often after the AND instruction that Intel added a separate instruction, TEST, whose main purpose was to logically AND two results and set the flags without otherwise affecting either instruction operand.

There are three main uses of the zero flag after the execution of an AND or TEST instruction: (1) checking to see if a particular bit in an operand is set; (2) checking to see if at least one of several bits in a bit set is one; and (3) checking to see if an operand is zero. Use (1) is actually a special case of (2) where the bit set contains only a single bit. We'll explore each of these uses in the following paragraphs.

A common use for the AND instruction, and also the original reason for the inclusion of the TEST instruction in the 80x86 instruction set, is to test to see if a particular bit is set in a given operand. To perform this type of test, you would normally AND/TEST a constant value containing a single set bit with the operand you wish to test. These clears all the other bits in the second operand leaving a zero in the bit position under test (the bit position with the single set bit in the constant operand) if the operand contains a zero in that position and leaving a one if the operand contains a one in that position. Since all of the other bits in the result are zero, the entire result will be zero if that particular bit is zero, the entire result will be non-zero if that bit position contains a one. The 80x86 reflects this status in the zero flag (Z=1 indicates a zero bit, Z=0 indicates a one bit). The following instruction sequence demonstrates how to test to see if bit four is set in EAX:

```
test( %1_000, eax );  // Check bit #4 to see if it is 0/1
if( @nz ) then

    << Do this if the bit is set >>
```

1. Serial communications chips and other communication hardware that uses parity for error checking normally computes the parity in hardware, you don't have to use software for this purpose.

```
        else

            << Do this if the bit is clear >>

        endif;
```

You can also use the AND/TEST instructions to see if any one of several bits is set. Simply supply a constant that has one bits in all the positions you want to test (and zeros everywhere else). ANDing such a value with an unknown quantity will produce a non-zero value if one or more of the bits in the operand under test contain a one. The following example tests to see if the value in EAX contains a one in bit positions one, two, four, and seven:

```
        test( %1001_0010, eax );
        if( @nz ) then // at least one of the bits is set.

            << do whatever needs to be done if one of the bits is set >>

        endif;
```

Note that you cannot use a single AND or TEST instruction to see if all the corresponding bits in the bit set are equal to one. To accomplish this, you must first mask out the bits that are not in the set and then compare the result against the mask itself. If the result is equal to the mask, then all the bits in the bit set contain ones. You must use the AND instruction for this operation as the TEST instruction does not mask out any bits. The following example checks to see if all the bits corresponding to a value this code calls *bitMask* are equal to one:

```
        and( bitMask, eax );
        cmp( eax, bitMask );
        if( @e ) then

            << All the bit positions in EAX corresponding to the set >>
            << bits in bitMask are equal to one if we get here.      >>

        endif;
```

Of course, once we stick the CMP instruction in there, we don't really have to check to see if all the bits in the bit set contain ones. We can check for any combination of values by specifying the appropriate value as the operand to the CMP instruction.

Note that the TEST/AND instructions will only set the zero flag in the above code sequences if all the bits in EAX (or other destination operand) have zeros in the positions where ones appear in the constant operand. This suggests another way to check for all ones in the bit set: invert the value in EAX prior to using the AND or TEST instruction. Then if the zero flag is set, you know that there were all ones in the (original) bit set, e.g.,

```
        not( eax );
        test( bitMask, eax );
        if( @z ) then

            << At this point, EAX contained all ones in the bit positions >>
            << occupied by ones in the bitMask constant.                  >>

        endif;
```

The paragraphs above all suggest that the *bitMask* (i.e., source operand) is a constant. This was for purposes of example only. In fact, you can use a variable or other register here, if you prefer. Simply load that variable or register with the appropriate bit mask before you execute the TEST, AND, or CMP instructions in the examples above.

Another set of instructions we've already seen that we can use to manipulate bits are the bit test instructions. These instructions include BT (bit test), BTS (bit test and set), BTC (bit test and complement), and

BTR (bit test and reset). We've used these instructions to manipulate bits in HLA character set variables, we can also use them to manipulate bits in general. The BT*x* instructions allow the following syntactical forms:

```
btx( BitNumber, BitsToTest );

btx( reg16, reg16 );
btx( reg32, reg32 );
btx( constant, reg16 );
btx( constant, reg32 );

btx( reg16, mem16 );
btx( reg32, mem32 );
btx( constant, mem16 );
btx( constant, mem32 );
```

The BT instruction's first operand is a bit number that specifies which bit to check in the second operand. If the second operand is a register, then the first operand must contain a value between zero and the size of the register (in bits) minus one; since the 80x86's largest registers are 32 bits, this value have the maximum value 31 (for 32-bit registers). If the second operand is a memory location, then the bit count is not limited to values in the range 0..31. If the first operand is a constant, it can be any eight-bit value in the range 0..255. If the first operand is a register, it has no limitation.

The BT instruction copies the specified bit from the second operand into the carry flag. For example, the "bt(8, eax);" instruction copies bit number eight of the EAX register into the carry flag. You can test the carry flag after this instruction to determine whether bit eight was set or clear in EAX.

In general, the BT instruction is, perhaps, not the best instruction for testing individual bits in a register. The TEST (or AND) instruction is a bit more efficient. These latter two instructions are Intel "RISC Core" instructions while the BT instruction is a "Complex" instruction. Therefore, you will often get better performance using TEST or AND rather than BT. If you want to test bits in memory operands (especially in bit arrays), then the BT instruction is probably a reasonable way to go.

The BTS, BTC, and BTR instructions manipulate the bit they test while they are testing it. These instructions are rather slow and you should avoid them if performance is your primary concern. In a later volume when we discuss semaphores you will see the true purpose for these instructions. Until then, if performance (versus convenience) is an issue, you should always try two different algorithms, one that uses these instructions, one that uses AND/OR instructions, and measure the performance difference; then choose the best of the two different approaches.

The shift and rotate instructions are another group of instructions you can use to manipulate and test bits. Of course, all of these instructions move the H.O. (left shift/rotate) or L.O. (right shift/rotate) bits into the carry flag. Therefore, you can test the carry flag after you execute one of these instructions to determine the original setting of the operand's H.O. or L.O. bit (depending on the direction). Of course, the shift and rotate instructions are invaluable for aligning bit strings, packing, and unpacking data. Volume One has several examples of this and, of course, some earlier examples in the section also use the shift instructions for this purpose.

5.4 The Carry Flag as a Bit Accumulator

The BT*x*, shift, and rotate instructions all set or clear the carry flag depending on the operation and/or selected bit. Since these instructions place their "bit result" in the carry flag, it is often convenient to think of the carry flag as a one-bit register or accumulator for bit operations. In this section we will explore some of the operations possible with this bit result in the carry flag.

Instructions that will be useful for manipulating bit results in the carry flag are those that use the carry flag as some sort of input value. The following is a sampling of such instructions:

- ADC, SBB
- RCL, RCR

- CMC (we'll throw in CLC and STC even though they don't use the carry as input)
- JC, JNC
- SETC, SETNC

The ADC and SBB instructions add or subtract their operands along with the carry flag. So if you've computed some bit result into the carry flag, you can figure that result into an addition or subtraction using these instructions. This isn't a common operation, but it is available if it's useful to you.

To merge a bit result in the carry flag, you most often use the rotate through carry instructions (RCL and RCR). These instructions, of course, move the carry flag into the L.O. or H.O. bits of their destination operand. These instructions are very useful for packing a set of bit results into a byte, word, or double word value.

The CMC (complement carry) instruction lets you easily invert the result of some bit operation. You can also use the CLC and STC instructions to initialize the carry flag prior to some string of bit operations involving the carry flag.

Of course, instructions that test the carry flag are going to be very popular after a calculation that leaves a bit result in the carry flag. The JC, JNC, SETC, and SETNC instructions are quite useful here. You can also use the HLA @C and @NC operands in a boolean expression to test the result in the carry flag.

If you have a sequence of bit calculations and you would like to test to see if the calculations produce a specific sequence of one-bit results, the easiest way to do this is to clear a register or memory location and use the RCL or RCR instructions to shift each result into that location. Once the bit operations are complete, then you can compare the register or memory location against a constant value to see if you've obtained a particular result sequence. If you want to test a sequence of results involving conjunction and disjunction (i.e., strings of results involving ANDs and ORs) then you could use the SETC and SETNC instruction to set a register to zero or one and then use the AND/OR instructions to merge the results.

5.5 Packing and Unpacking Bit Strings

A common bit operation is inserting a bit string into an operand or extracting a bit string from an operand. Previous chapters in this text have provided simple examples of packing and unpacking such data, now it is time to formally describe how to do this.

For the purposes of the current discussion, we will assume that we're dealing with bit strings; that is, a contiguous sequence of bits. A little later in this chapter we'll take a look at how to extract and insert bit sets in an operand. Another simplification we'll make is that the bit string completely fits within a byte, word, or double word operand. Large bit strings that cross object boundaries require additional processing; a discussion of bit strings that cross double word boundaries appears later in this section.

A bit string has two attributes that we must consider when packing and unpacking that bit string: a starting bit position and a length. The starting bit position is the bit number of the L.O. bit of the string in the larger operand. The length, of course, is the number of bits in the operand. To insert (pack) data into a destination operand we will assume that we start with a bit string of the appropriate length that is right-justified (i.e., starts in bit position zero) in an operand and is zero extended to eight, sixteen, or thirty-two bits. The task is to insert this data at the appropriate starting position in some other operand that is eight, sixteen, or thirty-bits wide. There is no guarantee that the destination bit positions contain any particular value.

The first two steps (which can occur in any order) is to clear out the corresponding bits in the destination operand and shift (a copy of) the bit string so that the L.O. bit begins at the appropriate bit position. After completing these two steps, the third step is to OR the shifted result with the destination operand. This inserts the bit string into the destination operand.

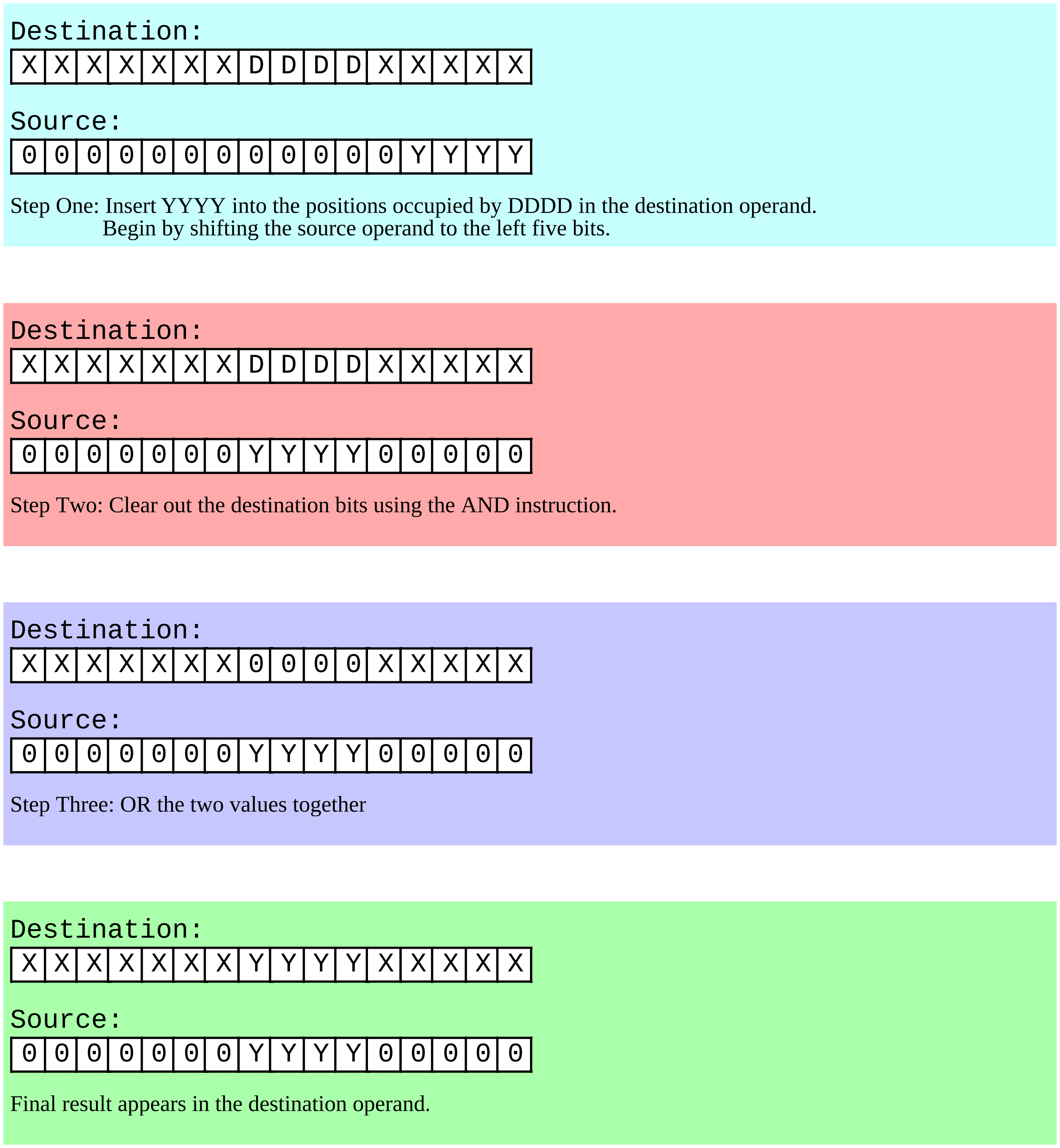

Destination:
X X X X X X X D D D D X X X X X
Source:
0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y
Step One: Insert YYYY into the positions occupied by DDDD in the destination operand.
Begin by shifting the source operand to the left five bits.
Destination:
X X X X X X X D D D D X X X X X
Source:
0 0 0 0 0 0 0 Y Y Y Y 0 0 0 0 0
Step Two: Clear out the destination bits using the AND instruction.
Destination:
X X X X X X X 0 0 0 0 X X X X X
Source:
0 0 0 0 0 0 0 Y Y Y Y 0 0 0 0 0
Step Three: OR the two values together
Destination:
X X X X X X X Y Y Y Y X X X X X
Source:
0 0 0 0 0 0 0 Y Y Y Y 0 0 0 0 0
Final result appears in the destination operand.

Figure 5.3 Inserting a Bit String Into a Destination Operand

It only takes three instructions to insert a bit string of known length into a destination operand. The following three instructions demonstrate how to handle the insertion operation in Figure 5.3; These instructions assume that the source operand is in BX and the destination operand is AX:

```
shl( 5, bx );
and( %111111000011111, ax );
or( bx, ax );
```

If the length and the starting position aren't known when you're writing the program (that is, you have to calculate them at run time), then bit string insertion is a little more difficult. However, with the use of a lookup table it's still an easy operation to accomplish. Let's assume that we have two eight-bit values: a starting bit position for the field we're inserting and a non-zero eight-bit length value. Also assume that the source operand is in EBX and the destination operand is EAX. The code to insert one operand into another could take the following form:

```
readonly

    // The index into the following table specifies the length of the bit string
    // at each position:

    MaskByLen: dword[ 32 ] :=
        [
            0,  $1,  $3,  $7, $f, $1f, $3f, $7f,
            $ff, $1ff, $3ff, $7ff, $fff, $1fff, $3fff, $7fff, $ffff,
            $1_ffff, $3_ffff, $7_ffff, $f_ffff,
            $1f_ffff, $3f_ffff, $7f_ffff, $ff_ffff,
            $1ff_ffff, $3ff_ffff, $7ff_ffff, $fff_ffff,
            $1fff_ffff, $3fff_ffff, $7fff_ffff, $ffff_ffff
        ];
            .
            .
            .
        movzx( Length, edx );
        mov( MaskByLen[ edx*4 ], edx );
        mov( StartingPosition, cl );
        shl( cl, edx );
        not( edx );
        shl( cl, ebx );
        and( edx, eax );
        or( ebx, eax );
```

Each entry in the *MaskByLen* table contains the number of one bits specified by the index into the table. Using the *Length* value as an index into this table fetches a value that has as many one bits as the *Length* value. The code above fetches an appropriate mask, shifts it to the left so that the L.O. bit of this run of ones matches the starting position of the field into which we want to insert the data, then it inverts the mask and uses the inverted value to clear the appropriate bits in the destination operand.

To extract a bit string from a larger operand is just as easy as inserting a bit string into some larger operand. All you've got to do is mask out the unwanted bits and then shift the result until the L.O. bit of the bit string is in bit zero of the destination operand. For example, to extract the four-bit field starting at bit position five in EBX and leave the result in EAX, you could use the following code:

```
        mov( ebx, eax );           // Copy data to destination.
        and( %1_1110_0000, ebx );  // Strip unwanted bits.
        shr( 5, eax );             // Right justify to bit position zero.
```

If you do not know the bit string's length and starting position when you're writing the program, you can still extract the desired bit string. The code is very similar to insertion (though a tiny bit simpler). Assuming you have the *Length* and *StartingPosition* values we used when inserting a bit string, you can extract the corresponding bit string using the following code (assuming source=EBX and dest=EAX):

```
        movzx( Length, edx );
        mov( MaskByLen[ edx*4 ], edx );
        mov( StartingPosition, cl );
        mov( ebx, eax );
        shr( cl, eax );
        and( edx, eax );
```

The examples up to this point all assume that the bit string appears completely within a double word (or smaller) object. This will always be the case if the bit string is less than or equal to 24 bits in length. However, if the length of the bit string plus its starting position (mod eight) within an object is greater than 32, then the bit string will cross a double word boundary within the object. To extract such bit strings requires up to three operations: one operation to extract the start of the bit string (up to the first double word boundary), an operation that copies whole double words (assuming the bit string is so long that it consumes several double words), and a final operation that copies left-over bits in the last double word at the end of the bit string. The actual implementation of this operation is left as an exercise at the end of this volume.

5.6 Coalescing Bit Sets and Distributing Bit Strings

Inserting and extracting bit sets is little different than inserting and extract bit strings if the “shape” of the bit set you’re inserting (or resulting bit set you’re extracting) is the same as the bit set in the main object. The “shape” of a bit set is the distribution of the bits in the set, ignoring the starting bit position of the set. So a bit set that includes bits zero, four, five, six, and seven has the same shape as a bit set that includes bits 12, 16, 17, 18, and 19 since the distribution of the bits is the same. The code to insert or extract this bit set is nearly identical to that of the previous section; the only difference is the mask value you use. For example, to insert this bit set starting at bit number zero in EAX into the corresponding bit set starting at position 12 in EBX, you could use the following code:

```
and( %1111_0001_0000_0000_0000, ebx );  // Mask out destination bits.
shl( 12, eax );                         // Move source bits into posn.
or( eax, ebx );                         // Merge the bit set into EBX.
```

However, suppose you have five bits in bit positions zero through four in EAX and you want to merge them into bits 12, 16, 17, 18, and 19 in EBX. Somehow you’ve got to distribute the bits in EAX prior to logically ORing the values into EBX. Given the fact that this particular bit set has only two runs of one bits, the process is somewhat simplified, the following code achieves this in a somewhat sneaky fashion:

```
and( %1111_0001_0000_0000_0000, ebx );
shl( 3, eax );   // Spread out the bits: 1-4 goes to 4-7 and 0 to 3.
btr( 3, eax );   // Bit 3->carry and then clear bit 3
rcl( 12, eax );  // Shift in carry and put bits into final position
or( eax, ebx );  // Merge the bit set into EBX.
```

This trick with the BTR (bit test and reset) instruction worked well because we only had one bit out of place in the original source operand. Alas, had the bits all been in the wrong location relative to one another, this scheme might not have worked quite as well. We’ll see a more general solution in just a moment.

Extracting this bit set and collecting (“coalescing”) the bits into a bit string is not quite as easy. However, there are still some sneaky tricks we can pull. Consider the following code that extracts the bit set from EBX and places the result into bits 0..4 of EAX:

```
mov( ebx, eax );
and( %1111_0001_0000_0000_0000, eax );  // Strip unwanted bits.
shr( 5, eax );                          // Put bit 12 into bit 7, etc.
shr( 3, ah );                           // Move bits 11..14 to 8..11.
shr( 7, eax );                          // Move down to bit zero.
```

This code moves (original) bit 12 into bit position seven, the H.O. bit of AL. At the same time it moves bits 16..19 down to bits 11..14 (bits 3..6 of AH). Then the code shifts the bits 3..6 in AH down to bit zero. This positions the H.O. bits of the bit set so that they are adjacent to the bit left in AL. Finally, the code shifts all the bits down to bit zero. Again, this is not a general solution, but it shows a clever way to attack this problem if you think about it carefully.

The problem with the coalescing and distribution algorithms above is that they are not general. They apply only to their specific bit sets. In general, specific solutions are going to provide the most efficient solution. A generalized solution (perhaps that lets you specify a mask and the code distributes or coalesces the bits accordingly) is going to be a bit more difficult. The following code demonstrates how to distribute the bits in a bit string according to the values in a bit mask:

```
//  EAX- Originally contains some value into which we insert bits from EBX.
//  EBX- L.O. bits contain the values to insert into EAX.
//  EDX- bitmap with ones indicating the bit positions in EAX to insert.
//  CL-  Scratchpad register.

            mov( 32, cl );    // Count # of bits we rotate.
            jmp DistLoop;

CopyToEAX:  rcr( 1, ebx );    // Don't use SHR here, must preserve Z-flag.
            rcr( 1, eax );
            jz  Done;
DistLoop:   dec( cl );
            shr( 1, edx );
            jc CopyToEAX;
            ror( 1, eax );    // Keep current bit in EAX.
            jnz DistLoop;

Done:       ror( cl, eax );   // Reposition remaining bits.
```

In the code above, if we load EDX with %1100_1001 then this code will copy bits 0..3 to bits 0, 3, 6, and 7 in EAX. Notice the short circuit test that checks to see if we've exhausted the values in EDX (by checking for a zero in EDX). Note that the rotate instructions do not affect the zero flag while the shift instructions do. Hence the SHR instruction above will set the zero flag when there are no more bits to distribute (i.e., when EDX becomes zero).

The general algorithm for coalescing bits is a tad more efficient than distribution. Here's the code that will extract bits from EBX via the bit mask in EDX and leave the result in EAX:

```
// EAX- Destination register.
// EBX- Source register.
// EDX- Bitmap with ones representing bits to copy to EAX.
// EBX and EDX are not preserved.

            sub( eax, eax );  // Clear destination register.
            jmp ShiftLoop;

ShiftInEAX: rcl( 1, ebx );    // Up here we need to copy a bit from
            rcl( 1, eax );    //  EBX to EAX.
ShiftLoop:  shl( 1, edx );    // Check mask to see if we need to copy a bit.
            jc ShiftInEAX;    // If carry set, go copy the bit.
            rcl( 1, ebx );    // Current bit is uninteresting, skip it.
            jnz ShiftLoop;    // Repeat as long as there are bits in EDX.
```

This sequence takes advantage of one sneaky trait of the shift and rotate instructions: the shift instructions affect the zero flag while the rotate instructions do not. Therefore, the "shl(1, edx);" instruction sets the zero flag when EDX becomes zero (after the shift). If the carry flag was also set, the code will make one additional pass through the loop in order to shift a bit into EAX, but the next time the code shifts EDX one bit to the left, EDX is still zero and so the carry will be clear. On this iteration, the code falls out of the loop.

Another way to coalesce bits is via table lookup. By grabbing a byte of data at a time (so your tables don't get too large) you can use that byte's value as an index into a lookup table that coalesces all the bits down to bit zero. Finally, you can merge the bits at the low end of each byte together. This might produce a more efficient coalescing algorithm in certain cases. The implementation is left to the reader...

5.7 Packed Arrays of Bit Strings

Although it is far more efficient to create arrays whose elements' have an integral number of bytes, it is quite possible to create arrays of elements whose size is not a multiple of eight bits. The drawback is that calculating the "address" of an array element and manipulating that array element involves a lot of extra work. In this section we'll take a look at a few examples of packing and unpacking array elements in an array whose elements are an arbitrary number of bits long.

Before proceeding, it's probably worthwhile to discuss why you would want to bother with arrays of bit objects. The answer is simple: space. If an object only consumes three bits, you can get 2.67 times as many elements into the same space if you pack the data rather than allocating a whole byte for each object. For very large arrays, this can be a substantial savings. Of course, the cost of this space savings is speed: you've got to execute extra instructions to pack and unpack the data, thus slowing down access to the data.

The calculation for locating the bit offset of an array element in a large block of bits is almost identical to the standard array access; it is

Element_Address_in_bits = Base_address_in_bits + index * element_size_in_bits

Once you calculate the element's address in bits, you need to convert it to a byte address (since we have to use byte addresses when accessing memory) and extract the specified element. Because the base address of an array element (almost) always starts on a byte boundary, we can use the following equations to simplify this task:

```
Byte_of_1st_bit = Base_Address + (index * element_size_in_bits )/8
Offset_to_1st_bit = (index * element_size_in_bits) % 8    (note "%" = MOD)
```

For example, suppose we have an array of 200 three-bit objects that we declare as follows:

```
static
    AO3Bobjects: byte[ (200*3)/8 + 1 ];  // "+1" handles trucation.
```

The constant expression in the dimension above reserves space for enough bytes to hold 600 bits (200 elements, each three bits long). As the comment notes, the expression adds an extra byte at the end to ensure we don't lose any odd bits (that won't happen in this example since 600 is evenly divisible by 8, but in general you can't count on this; one extra byte usually won't hurt things).

Now suppose you want to access the ith three-bit element of this array. You can extract these bits by using the following code:

```
// Extract the ith group of three bits in AO3Bobjects and leave this value
// in EAX.

        sub( ecx, ecx );      // Put i/8 remainder here.
        mov( i, eax );        // Get the index into the array.
        shrd( 3, eax, ecx );  // EAX/8 -> EAX and EAX mod 8 -> ECX (H.O. bits)
        shr( 3, eax );        // Remember, shrd above doesn't modify eax.
        rol( 3, ecx );        // Put remainder into L.O. three bits of ECX.

        // Okay, fetch the word containing the three bits we want to extract.
        // We have to fetch a word because the last bit or two could wind up
        // crossing the byte boundary (i.e., bit offset six and seven in the
        // byte).

        mov( AO3Bobjecs[eax], eax );
        shr( cl, eax );       // Move bits down to bit zero.
        and( %111, eax );     // Remove the other bits.
```

Inserting an element into the array is a bit more difficult. In addition to computing the base address and bit offset of the array element, you've also got to create a mask to clear out the bits in the destination where you're going to insert the new data. The following code inserts the L.O. three bits of EAX into the *ith* element of the *AO3Bobjects* array.

```
// Insert the L.O. three bits of AX into the ith element of AO3Bobjects:

readonly
   Masks: word[8] :=
         [
            !%0000_0111,  !%0000_1110,  !%0001_1100, !%0011_1000,
            !%0111_0000,  !%1110_0000,  !%1_1100_0000, !%11_1000_0000
         ];
            .
            .
            .
      sub( ecx, ecx );      // Put remainder here.
      mov( i, ebx );        // Get the index into the array.
      shrd( 3, ebx, ecx );  // i/8 -> EBX, i % 8 -> ECX.
      shr( 3, ebx );
      rol( 3, ecx );

      and( %111, ax );                // Clear unneeded bits from AX.
      mov( Masks[ecx], dx );          // Mask to clear out our array element.
      and( AO3Bobjects[ ebx ], dx );  // Grab the bits and clear those
                                      // we're inserting.
      shl( cl, ax );       // Put our three bits in their proper location.
      or( ax, dx );        // Merge bits into destination.
      mov( dx, AO3Bobjects[ ebx ] );  // Store back into  memory.
```

Notice the use of a lookup table to generate the masks needed to clear out the appropriate position in the array. Each element of this array contains all ones except for three zeros in the position we need to clear for a given bit offset (note the use of the "!" operator to invert the constants in the table).

5.8 Searching for a Bit

A very common bit operation is to locate the end of some run of bits. A very common special case of this operation is to locate the first (or last) set or clear bit in a 16- or 32-bit value. In this section we'll explore ways to accomplish this.

Before describing how to search for the first or last bit of a given value, perhaps it's wise to discuss exactly what the terms "first" and "last" mean in this context. The term "first set bit" means the first bit in a value, scanning from bit zero towards the high order bit, that contains a one. A similar definition exists for the "first clear bit." The "last set bit" is the first bit in a value, scanning from the high order bit towards bit zero, that contains a one. A similar definition exists for the last clear bit.

One obvious way to scan for the first or last bit is to use a shift instruction in a loop and count the number of iterations before you shift out a one (or zero) into the carry flag. The number of iterations specifies the position. Here's some sample code that checks for the first set bit in EAX and returns that bit position in ECX:

```
       mov( -32, ecx );      // Count off the bit positions in ECX.
TstLp: shr( 1, eax );        // Check to see if current bit position contains
       jc Done               //   a one;  exit loop if it does.
       inc( ecx );           // Bump up our bit counter by one.
       jnz TstLp;            // Exit if we execute this loop 32 times.

Done:  add( 32, cl );        // Adjust loop counter so it holds the bit posn.

// At this point, ECX contains the bit position of the first set bit.
// ECX contains 32 if EAX originally contained zero (no set bits).
```

The only thing tricky about this code is the fact that it runs the loop counter from -32 to zero rather than 32 down to zero. This makes it slightly easier to calculate the bit position once the loop terminates.

The drawback to this particular loop is that it's expensive. This loop repeats as many as 32 times depending on the original value in EAX. If the values you're checking often have lots of zeros in the L.O. bits of EAX, this code runs rather slow.

Searching for the first (or last) set bit is such a common operation that Intel added a couple of instructions on the 80386 specifically to accelerate this process. These instructions are BSF (bit scan forward) and BSR (bit scan reverse). Their syntax is as follows:

```
bsr( source, destReg );
bsf( source, destReg );
```

The source and destinations operands must be the same size and they must both be 16- or 32-bit objects. The destination operand has to be a register, the source operand can be a register or a memory location.

The BSF instruction scans for the first set bit (starting from bit position zero) in the source operand. The BSR instruction scans for the last set bit in the source operand by scanning from the H.O. bit towards the L.O. bit. If these instructions find a bit that is set in the source operand then they clear the zero flag and put the bit position into the destination register. If the source register contains zero (i.e., there are no set bits) then these instructions set the zero flag and leave an indeterminate value in the destination register. Note that you should test the zero flag immediately after the execution of these instructions to validate the destination register's value. Examples:

```
mov( SomeValue, ebx );      // Value whose bits we want to check.
bsf( ebx. eax );            // Put position of first set bit in EAX.
jz NoBitsSet;               // Branch if SomeValue contains zero.
mov( eax, FirstBit );       // Save location of first set bit.
 .
 .
 .
```

You use the BSR instruction in an identical fashion except that it computes the bit position of the last set bit in an operand (that is, the first set bit it finds when scanning from the H.O. bit towards the L.O. bit).

The 80x86 CPUs do not provide instructions to locate the first bit containing a zero. However, you can easily scan for a zero bit by first inverting the source operand (or a copy of the source operand if you must preserve the source operand's value). If you invert the source operand, then the first "1" bit you find corresponds to the first zero bit in the original operand value.

The BSF and BSR instructions are complex instructions (i.e., they are not a part of the 80x86 "RISC core" instruction set). Therefore, these instructions are necessarily as fast as other instructions. Indeed, in some circumstances it may be faster to locate the first set bit using discrete instructions. However, since the execution time of these instructions varies widely from CPU to CPU, you should first test the performance of these instructions prior to using them in time critical code.

Note that the BSF and BSR instructions do not affect the source operand. A common operation is to extract the first (or last) set bit you find in some operand. That is, you might want to clear the bit once you find it. If the source operand is a register (or you can easily move it into a register) then you can use the BTR (or BTC) instruction to clear the bit once you've found it. Here's some code that achieves this result:

```
bsf( eax, ecx );     // Locate first set bit in EAX.
if( @nz ) then       // If we found a bit, clear it.

    btr( ecx, eax ); // Clear the bit we just found.

endif;
```

At the end of this sequence, the zero flag indicates whether we found a bit (note that BTR does not affect the zero flag). Alternately, you could add an ELSE section to the IF statement above that handles the case when the source operand (EAX) contains zero at the beginning of this instruction sequence.

Since the BSF and BSR instructions only support 16- and 32-bit operands, you will have to compute the first bit position of an eight-bit operand a little differently. There are a couple of reasonable approaches. First, of course, you can usually zero extend an eight-bit operand to 16 or 32 bits and then use the BSF or

BSR instructions on this operand. Another alternative is to create a lookup table where each entry in the table contains the number of bits in the value you use as an index into the table; then you can use the XLAT instruction to "compute" the first bit position in the value (note that you will have to handle the value zero as a special case). Another solution is to use the shift algorithm appearing at the beginning of this section; for an eight-bit operand, this is not an entirely inefficient solution.

One interesting use of the BSF and BSR instructions is to "fill in" a character set with all the values from the lowest-valued character in the set through the highest-valued character. For example, suppose a character set contains the values {'A', 'M', 'a'..'n', 'z'}; if we filled in the gaps in this character set we would have the values {'A'..'z'}. To compute this new set we can use BSF to determine the ASCII code of the first character in the set and BSR to determine the ASCII code of the last character in the set. After doing this, we can feed those two ASCII codes to the *cs.rangeChar* function to compute the new set.

You can also use the BSF and BSR instructions to determine the size of a run of bits, assuming that you have a single run of bits in your operand. Simply locate the first and last bits in the run (as above) and the compute the difference (plus one) of the two values. Of course, this scheme is only valid if there are no intervening zeros between the first and last set bits in the value.

5.9 Counting Bits

The last example in the previous section demonstrates a specific case of a very general problem: counting bits. Unfortunately, that example has a severe limitation: it only counts a single run of one bits appearing in the source operand. This section discusses a more general solution to this problem.

Hardly a week goes by that someone doesn't ask how to count the number of bits in a register operand on one of the Internet news groups. This is a common request, undoubtedly, because many assembly language course instructors assign this task a project to their students as a way to teach them about the shift and rotate instructions. Undoubtedly, the solution these instructor expect is something like the following:

```
// BitCount1:
//
//   Counts the bits in the EAX register, returning the count in EBX.

          mov( 32, cl );     // Count the 32 bits in EAX.
          sub( ebx, ebx );   // Accumulate the count here.
CntLoop:  shr( 1, eax );     // Shift next bit out of EAX and into Carry.
          adc( 0, bl );      // Add the carry into the EBX register.
          dec( cl );         // Repeat 32 times.
          jnz CntLoop
```

The "trick" worth noting here is that this code uses the ADC instruction to add the value of the carry flag into the BL register. Since the count is going to be less than 32, the result will fit comfortably into BL. This code uses "adc(0, bl);" rather than "adc(0, ebx);" because the former instruction is smaller.

Tricky code or not, this instruction sequence is not particularly fast. As you can tell with just a small amount of analysis, the loop above always executes 32 times, so this code sequence executes 130 instructions (four instructions per iteration plus two extra instructions). One might ask if there is a more efficient solution, the answer is yes. The following code, taken from the AMD Athlon optimization guide, provides a faster solution (see the comments for a description of the algorithm):

```
    // bitCount-
    //
    //  Counts the number of "1" bits in a dword value.
    //  This function returns the dword count value in EAX.

    procedure bits.cnt( BitsToCnt:dword ); nodisplay;

    const
        EveryOtherBit       := $5555_5555;
        EveryAlternatePair  := $3333_3333;
        EvenNibbles         := $0f0f_0f0f;

    begin cnt;

        push( edx );
        mov( BitsToCnt, eax );
        mov( eax, edx );

        // Compute sum of each pair of bits
        // in EAX.  The algorithm treats
        // each pair of bits in EAX as a two
        // bit number and calculates the
        // number of bits as follows (description
        // is for bits zero and one, it generalizes
        // to each pair):
        //
        //  EDX =   BIT1  BIT0
        //  EAX =      0  BIT1
        //
        //  EDX-EAX =   00 if both bits were zero.
        //              01 if Bit0=1 and Bit1=0.
        //              01 if Bit0=0 and Bit1=1.
        //              10 if Bit0=1 and Bit1=1.
        //
        // Note that the result is left in EDX.

        shr( 1, eax );
        and( EveryOtherBit, eax );
        sub( eax, edx );

        // Now sum up the groups of two bits to
        // produces sums of four bits.  This works
        // as follows:
        //
        //  EDX = bits 2,3, 6,7, 10,11, 14,15, ..., 30,31
        //        in bit positions 0,1, 4,5, ..., 28,29 with
        //        zeros in the other positions.
        //
        //  EAX = bits 0,1, 4,5, 8,9, ... 28,29 with zeros
        //        in the other positions.
        //
        //  EDX+EAX produces the sums of these pairs of bits.
        //  The sums consume bits 0,1,2, 4,5,6, 8,9,10, ... 28,29,30
        //  in EAX with the remaining bits all containing zero.

        mov( edx, eax );
        shr( 2, edx );
        and( EveryAlternatePair, eax );
        and( EveryAlternatePair, edx );
        add( edx, eax );

        // Now compute the sums of the even and odd nibbles in the
        // number.  Since bits 3, 7, 11, etc. in EAX all contain
        // zero from the above calcuation, we don't need to AND
        // anything first, just shift and add the two values.
        // This computes the sum of the bits in the four bytes
        // as four separate value in EAX (AL contains number of
        // bits in original AL, AH contains number of bits in
        // original AH, etc.)
        mov( eax, edx );
        shr( 4, eax );
        add( edx, eax );
        and( EvenNibbles, eax );

        // Now for the tricky part.
        // We want to compute the sum of the four bytes
        // and return the result in EAX.  The following
        // multiplication achieves this.  It works
        // as follows:
        //  (1) the $01 component leaves bits 24..31
        //      in bits 24..31.
        //
        //  (2) the $100 component adds bits 17..23
        //      into bits 24..31.
        //
        //  (3) the $1_0000 component adds bits 8..15
        //      into bits 24..31.
        //
        //  (4) the $1000_0000 component adds bits 0..7
        //      into bits 24..31.
        //
        //  Bits 0..23 are filled with garbage, but bits
        //  24..31 contain the actual sum of the bits
        //  in EAX's original value.  The SHR instruction
        //  moves this value into bits 0..7 and zeroes
        //  out the H.O. bits of EAX.

        intmul( $0101_0101, eax );
        shr( 24, eax );

        pop( edx );

    end cnt;
```

5.10 Reversing a Bit String

Another common programming project instructions assign, and a useful function in its own right, is a program that reverses the bits in an operand. That is, it swaps the L.O. bit with the H.O. bit, bit #1 with the next-to-H.O. bit, etc. The typical solution an instructor probably expects for this assignment is the following:

```
// Reverse the 32-bits in EAX, leaving the result in EBX:

          mov( 32, cl );
RvsLoop:  shr( 1, eax );    // Move current bit in EAX to the carry flag.
          rcl( 1, ebx );    // Shift the bit back into EBX, backwards.
          dec( cl );
          jnz RvsLoop
```

As with the previous examples, this code suffers from the fact that it repeats the loop 32 times for a grand total of 129 instructions. By unrolling the loop you can get it down to 64 instructions, but this is still somewhat expensive.

As usual, the best solution to an optimization problem is often a better algorithm rather than attempting to tweak your code by trying to choose faster instructions to speed up some code. However, a little intelligence goes a long way when manipulating bits. In the last section, for example, we were able to speed up counting the bits in a string by substituting a more complex algorithm for the simplistic "shift and count" algorithm. In the example above, we are once again faced with a very simple algorithm with a loop that repeats for one bit in each number. The question is: "Can we discover an algorithm that doesn't execute 129 instructions to reverse the bits in a 32-bit register?" The answer is "yes" and the trick is to do as much work as possible in parallel.

Suppose that all we wanted to do was swap the even and odd bits in a 32-bit value. We can easily swap the even an odd bits in EAX using the following code:

```
mov( eax, edx );         // Make a copy of the odd bits in the data.
shr( 1, eax );           // Move the even bits to the odd positions.
and( $5555_5555, edx );  // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax );  // Isolate the even bits (in odd posn now).
shl( 1, edx );           // Move the odd bits to the even positions.
or( edx, eax );          // Merge the bits and complete the swap.
```

Of course, swapping the even and odd bits, while somewhat interesting, does not solve our larger problem of reversing all the bits in the number. But it does take us part of the way there. For example, if after executing the code sequence above, we swap adjacent pairs of bits, then we've managed to swap the bits in all the nibbles in the 32-bit value. Swapping adjacent pairs of bits is done in a manner very similar to the above, the code is

```
mov( eax, edx );         // Make a copy of the odd numbered bit pairs.
shr( 2, eax );           // Move the even bit pairs to the odd posn.
and( $3333_3333, edx );  // Isolate the odd pairs by clearing even pairs.
and( $3333_3333, eax );  // Isolate the even pairs (in odd posn now).
shl( 2, edx );           // Move the odd pairs to the even positions.
or( edx, eax );          // Merge the bits and complete the swap.
```

After completing the sequence above we swap the adjacent nibbles in the 32-bit register. Again, the only difference is the bit mask and the length of the shifts. Here's the code:

```
mov( eax, edx );         // Make a copy of the odd numbered nibbles.
shr( 4, eax );           // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx );  // Isolate the odd nibbles.
and( $0f0f_0f0f, eax );  // Isolate the even nibbles (in odd posn now).
shl( 4, edx );           // Move the odd pairs to the even positions.
or( edx, eax );          // Merge the bits and complete the swap.
```

You can probably see the pattern developing and can figure out that in the next two steps we've got to swap the bytes and then the words in this object. You can use code like the above, but there is a better way – use the BSWAP instruction. The BSWAP (byte swap) instruction uses the following syntax:

bswap(reg$_{32}$);

This instruction swaps bytes zero and three and it swaps bytes one and two in the specified 32-bit register. The principle use of this instruction is to convert data between the so-called "little endian" and "big-endian" data formats[2]. Although we don't specifically need this instruction for this purpose here, the BSWAP instruction does swap the bytes and words in a 32-bit object exactly the way we want them when reversing bits, so rather than sticking in another 12 instructions to swap the bytes and then the words, we can simply use a "bswap(eax);" instruction to complete the job after the instructions above. The final code sequence is

```
mov( eax, edx );         // Make a copy of the odd bits in the data.
shr( 1, eax );           // Move the even bits to the odd positions.
and( $5555_5555, edx );  // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax );  // Isolate the even bits (in odd posn now).
shl( 1, edx );           // Move the odd bits to the even positions.
or( edx, eax );          // Merge the bits and complete the swap.
```

2. In the little endian system, which the native 80x86 format, the L.O. byte of an object appears at the lowest address in memory. In the big endian system, which various RISC processors use, the H.O. byte of an object appears at the lowest address in memory. The BSWAP instruction converts between these two data formats.

```
        mov( eax, edx );         // Make a copy of the odd numbered bit pairs.
        shr( 2, eax );           // Move the even bit pairs to the odd posn.
        and( $3333_3333, edx );  // Isolate the odd pairs by clearing even pairs.
        and( $3333_3333, eax );  // Isolate the even pairs (in odd posn now).
        shl( 2, edx );           // Move the odd pairs to the even positions.
        or( edx, eax );          // Merge the bits and complete the swap.

        mov( eax, edx );         // Make a copy of the odd numbered nibbles.
        shr( 4, eax );           // Move the even nibbles to the odd position.
        and( $0f0f_0f0f, edx );  // Isolate the odd nibbles.
        and( $0f0f_0f0f, eax );  // Isolate the even nibbles (in odd posn now).
        shl( 4, edx );           // Move the odd pairs to the even positions.
        or( edx, eax );          // Merge the bits and complete the swap.

        bswap( eax );            // Swap the bytes and words.
```

This algorithm only requires 19 instructions and it executes much faster than the bit shifting loop appearing earlier. Of course, this sequence does consume a bit more memory, so if you're trying to save memory rather than clock cycles, the loop is probably a better solution.

5.11 Merging Bit Strings

Another common bit string operation is producing a single bit string by merging, or interleaving, bits from two different sources. The following example code sequence creates a 32-bit string by merging alternate bits from two 16-bit strings:

```
// Merge two 16-bit strings into a single 32-bit string.
// AX - Source for even numbered bits.
// BX - Source for odd numbered bits.
// CL - Scratch register.
// EDX- Destination register.

           mov( 16, cl );
MergeLp:   shrd( 1, eax, edx );   // Shift a bit from EAX into EDX.
           shrd( 1, ebx, edx );   // Shift a bit from EBX into EDX.
           dec( cl );
           jne MergeLp;
```

This particular example merged two 16-bit values together, alternating their bits in the result value. For a faster implementation of this code, unrolling the loop is probably you're best bet since this eliminates half the instructions that execute on each iteration of the loop above.

With a few slight modifications, we could also have merged four eight-bit values together, or we could have generated the result using other bit sequences; for example, the following code copies bits 0..5 from EAX, then bits 0..4 from EBX, then bits 6..11 from EAX, then bits 5..15 from EBX, and finally bits 12..15 from EAX:

```
           shrd( 6, eax, edx );
           shrd( 5, ebx, edx );
           shrd( 6, eax, edx );
           shrd( 11, ebx, edx );
           shrd( 4, eax, edx );
```

5.12 Extracting Bit Strings

Of course, we can easily accomplish the converse of merging two bit streams; i.e., we can extract and distribute bits in a bit string among multiple destinations. The following code takes the 32-bit value in EAX and distributes alternate bits among the BX and DX registers:

```
              mov( 16, cl );        // Count off the number of loop iterations.
ExtractLp:‘   shr( 1, eax );        // Extract even bits to (E)BX.
              rcr( 1, ebx );
              shr( 1, eax );        // Extract odd bits to (E)DX.
              rcr( 1, edx );
              dec( cl );            // Repeat 16 times.
              jnz ExtractLp;
              shr( 16, ebx );       // Need to move the results from the H.O.
              shr( 16, edx );       //  bytes of EBX/EDX to the L.O. bytes.
```

This sequence executes 99 instructions. This isn’t terrible, but we can probably do a little better by using a better algorithm that extracts bits in parallel. Employing the technique we used to reverse bits in a register, we can come up with the following algorithm that relocates all the even bits to the L.O. word of EAX and all the odd bits to the H.O. word of EAX.

```
// Swap bits at positions (1,2), (5,6), (9,10), (13,14), (17,18),
// (21,22), (25,26), and (29, 30).

       mov( eax, edx );
       and( $9999_9999, eax );      // Mask out the bits we’ll keep for now.
       mov( edx, ecx );
       shr( 1, edx );               // Move 1st bits in tuple above to the
       and( $2222_2222, ecx );      //  correct position and mask out the
       and( $2222_2222, edx );      //  unneeded bits.
       shl( 1, ecx );               // Move 2nd bits in tuples above.
       or( edx, ecx );              // Merge all the bits back together.
       or( ecx, eax );

// Swap bit pairs at positions ((2,3), (4,5)),  ((10,11), (12,13)), etc.

       mov( eax, edx );
       and( $c3c3_c3c3, eax );      // The bits we’ll leave alone.
       mov( edx, ecx );
       shr( 2, edx );
       and( $0c0c_0c0c, ecx );
       and( $0c0c_0c0c, edx );
       shl( 2, ecx );
       or( edx, ecx );
       or( ecx, eax );

// Swap nibbles at nibble positions (1,2), (5,6), (9,10), etc.

       mov( eax, edx );
       and( $f00f_f00f, eax );
       mov( edx, ecx );
       shr(4, edx );
       and( $0f0f_0f0f, ecx );
       and( $0f0f_0f0f, ecx );
       shl( 4, ecx );
       or( edx, ecx );
       or( ecx, eax );

// Swap bits at positions 1 and 2.
```

```
        ror( 8, eax );
        xchg( al, ah );
        rol( 8, eax );
```

This sequence require 30 instructions. At first blush it looks like a winner since the original loop executes 64 instructions. However, this code isn't quite as good as it looks. After all, if we're willing to write this much code, why not unroll the loop above 16 times? That sequence only requires 64 instructions. So the complexity of the previous algorithm may not gain much on instruction count. As to which sequence is faster, well, you'll have to time them to figure this out. However, the SHRD instructions are not particularly fast, neither are the instructions in the other sequence. This example does not appear here to show you a better algorithm, but rather to demonstrate that writing really tricky code doesn't always provide a big performance boost.

Extracting other bit combinations is left as an exercise for the reader.

5.13 Searching for a Bit Pattern

Another bit-related operation you may need is the ability to search for a particular bit pattern in a string of bits. For example, you might want to locate the bit index of the first occurrence of %1011 starting at some particular position in a bit string. In this section we'll explore some simple algorithms to accomplish this task.

To search for a particular bit pattern we're going to need to know four things: (1) the pattern to search for (the *pattern*), (2) the length of the pattern we're searching for, (3) the bit string that we're going to search through (the *source*), and (4) the length of the bit string to search through. The basic idea behind the search is to create a mask based on the length of the pattern and mask a copy of the source with this value. Then we can directly compare the pattern with the masked source for equality. If they are equal, we're done; if they're not equal, then increment a bit position counter, shift the source one position to the right, and try again. We repeat this operation *length(source)* - *length(pattern)* times. The algorithm fails if it does not detect the bit pattern after this many attempts (since we will have exhausted all the bits in the source operand that could match the pattern's length). Here's a simple algorithm that searches for a four-bit pattern throughout the EBX register:

```
          mov( 28, cl );      // 28 attempts since 32-4 = 28 (len(src)-len(pat)).
          mov( %1111, ch );   // Mask for the comparison.
          mov( pattern, al ); // Pattern to search for.
          and( ch, al );      // Mask unnecessary bits in AL.
          mov( source, ebx ); // Get the source value.
ScanLp:   mov( bl, dl );      // Make a copy of the L.O. four bits of EBX
          and( ch, dl );      // Mask unwanted bits.
          cmp( dl, al );      // See if we match the pattern.
          jz Matched;
          dec( cl );          // Repeat the specified number of times.
          jnz ScanLp;

    << If we get to this point, we failed to match the bit string >>

          jmp Done;

Matched:
    << If we get to this point, we matched the bit string.  We can >>
    << compute the position in the original source as 28-cl.       >>

Done:
```

Bit string scanning is a special case of string matching. String matching is a well studied problem in Computer Science and many of the algorithms you can use for string matching are applicable to bit string matching as well. Such algorithms are a bit beyond the scope of this chapter, but to give you a preview of how this works, you compute some function (like XOR or SUB) between the pattern and the current source bits and use the result as an index into a lookup table to determine how many bits you can skip. Such algorithms let you skip several bits rather than only shifting once per each iteration of the scanning loop (as is done by the algorithm above). For more details on string scanning and their possible application to bit string matching, see the appropriate chapter in the volume on Advanced String Handling.

5.14 The HLA Standard Library Bits Module

The HLA Standard Library provides a “bits” module that provides several bit related functions, including built-in functions for many of the algorithms we’ve studied in this chapter. This section will describe these functions available in the HLA Standard Library.

```
procedure bits.cnt( b:dword ); returns( "EAX" );
```

This procedure returns the number of one bits present in the “b” parameter. It returns the count in the EAX register. To count the number of zero bits in the parameter value, invert the value of the parameter before passing it to *bits.cnt*. If you want to count the number of bits in a 16-bit operand, simply zero extend it to 32 bits prior to calling this function. Here are a couple of examples:

```
// Compute the number of bits in a 16-bit register:

        pushw( 0 );
        push( ax );
        call bits.cnt;

// If you prefer to use a higher-level syntax, try the following:

        bits.cnt( #{ pushw(0); push(ax); }# );

// Compute the number of bits in a 16-bit memory location:

        pushw( 0 );
        push( mem16 );
        bits.cnt;
```

If you want to compute the number of bits in an eight-bit operand it’s probably faster to write a simple loop that rotates all the bits in the source operand and adds the carry into the accumulating sum. Of course, if performance isn’t an issue, you can zero extend the byte to 32 bits and call the *bits.cnt* procedure.

```
procedure bits.distribute( source:dword; mask:dword; dest:dword );
    returns( "EAX" );
```

This function takes the L.O. *n* bits of *source*, where *n* is the number of “1” bits in *mask*, and inserts these bits into *dest* at the bit positions specified by the “1” bits in *mask* (i.e., the same as the distribute algorithm appearing earlier in this chapter). This function does not change the bits in *dest* that correspond to the zeros in the *mask* value. This function does not affect the value of the actual *dest* parameter, instead, it returns the new value in the EAX register.

```
procedure bits.coalese( source:dword; mask:dword );
    returns( "EAX" );
```

This function is the converse of *bits.distribute*. It extracts all the bits in source whose corresponding positions in mask contain a one. This function coalesces (right justifies) these bits in the L.O. bit positions of the result and returns the result in EAX.

```
procedure bits.extract( var d:dword ); returns( "EAX" );  // Really a macro.
```

This function extracts the first set bit in *d* searching from bit #0 and returns the index of this bit in the EAX register; the function will also return the zero flag clear in this case. This function also clears that bit in the operand. If *d* contains zero, then this function returns the zero flag set and EAX will contain -1.

Note that HLA actually implements this function as a macro, not a procedure (see the chapter on Macros for details). This means that you can pass any double word operand as a parameter (i.e., a memory or a register operand). However, the results are undefined if you pass EAX as the parameter (since this function computes the bit number in EAX).

```
procedure bits.reverse32( d:dword ); returns( "EAX" );
procedure bits.reverse16( w:word ); returns( "AX" );
procedure bits.reverse8( b:byte ); returns( "AL" );
```

These three routines return their parameter value with the its bits reversed in the accumulator register (AL/AX/EAX). Call the routine appropriate for your data size.

```
procedure bits.merge32( even:dword; odd:dword ); returns( "EDX:EAX" );
procedure bits.merge16( even:word; odd:word ); returns( "EAX" );
procedure bits.merge8( even:byte; odd:byte ); returns( "AX" );
```

These routines merge two streams of bits to produce a value whose size is the combination of the two parameters. The bits from the “even” parameter occupy the even bits in the result, the bits from the “odd” parameter occupy the odd bits in the result. Notice that these functions return 16, 32, or 64 bits based on byte, word, and double word parameter values.

```
procedure bits.nibbles32( d:dword ); returns( "EDX:EAX" );
procedure bits.nibbles16( w:word ); returns( "EAX" );
procedure bits.nibbles8( b:byte ); returns( "AX" );
```

These routines extract each nibble from the parameter and place those nibbles into individual bytes. The *bits.nibbles8* function extracts the two nibbles from the *b* parameter and places the L.O. nibble in AL and the H.O. nibble in AH. The *bits.nibbles16* function extracts the four nibbles in *w* and places them in each of the four bytes of EAX. You can use the BSWAP or ROx instructions to gain access to the nibbles in the H.O. word of EAX. The *bits.nibbles32* function extracts the eight nibbles in EAX and distributes them through the eight bytes in EDX:EAX. Nibble zero winds up in AL and nibble seven winds up in the H.O. byte of EDX. Again, you can use BSWAP or the rotate instructions to access the upper bytes of EAX and EDX.

5.15 Putting It All Together

Bit manipulation is one area where assembly language really shines. Not only is bit manipulation far more efficient in assembly language than in high level languages, but it’s often easier as well. Although the need to manipulate bits is not an everyday requirement, bit manipulation is still a very important problem area. In this chapter we’ve explored several ways to manipulate data as bits. Although this chapter only begins to cover the possibilities, it should give you some ideas for developing your own bit manipulation algorithms for use in your applications.