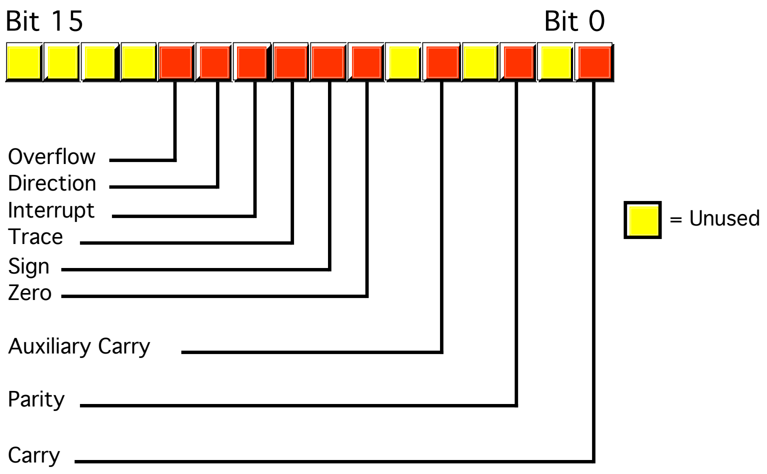
The 80x86 Instruction Set Lab Manual, Chapter Six

The Intel 80x86 processor family supports a wide variation of machine instructions. You can perform many operations with several different instruction sequences. While this variety makes it easier to develop an algorithm in assembly language, it complicates determining the *optimal* sequence of instructions necessary for the job. In this laboratory you will study the operation of many instructions in the 80x86 instruction set and compare instruction timings for sequences which perform the same operation.

6.1 The 80x86 Flags Register



8086 Flags

The 80x86 microprocessors use several bits in the flags register to control the operation of the CPU and denote the current machine state. The *condition code bits* (overflow, sign, zero, carry, and parity) let you test the results of completed computations. The machine state bits (direction, interrupt, and trace) let you control the operation of the machine. Finally the auxiliary carry flag holds important status information for BCD operations.

The 80x86 provides special instructions which let you test the settings of the five condition code bits. These instructions include the conditional jump (Jcc) instructions and the conditional set instructions (SETcc, available only on 80386 and later processors). Indeed, the whole reason these five flags have the name "condition codes" is because you can execute instructions which conditionally test them.

6.1 Which flags are the condition codes?

____

The machine state flags control the operation of the CPU. These flags include the direction flag, the trace flag, and the interrupt disable flag. The trace flag gives debuggers such as CodeView the ability to *single-step* through a section of code. Application programs rarely change this flag. The direction flag controls the operation of the string instructions. The interrupt disable flag controls whether hardware *interrupts* are active. Usually, interrupts are always on. An application can turn off interrupts for brief periods of time when executing certain critical code.

The auxiliary carry flag, set by certain arithmetic operations, denotes a carry while performing BCD arithmetic. You cannot directly set or clear it, nor can you directly test it. Since the text does not consider BCD arithmetic, we will ignore this flag.

The 80x86 provides several instructions which directly affect the flags. These include CLC (clear carry), STC (set carry), CMC (complement carry), CLD (clear direction), STD (set direction), CLI (clear interrupt), STI (set interrupt), POPF/POPFD (pop flags), and SAHF (store AH into flags).

Although the 80x86 does not provide specific instructions to set and clear other flags, you can use the LAHF/SAHF and PUSHF/POPF instructions to accomplish this. For example, to set the parity flag use an instruction sequence like

```
lahf
or     ah, 100b             ;Parity is bit #2 in the flags register.
sahf
```

Keep in mind that the SAHF instruction does not affect the overflow flag and the OR instruction clears the overflow flag. If you need to preserve or modify the overflow, interrupt, or trace flag in the above sequence, you will need to use code like the following:

```
pushf
pop    ax
or     ax, flag_bits        ;Use AND to clear bits.
push   ax
popf
```

6.2 Which flags do the SAHF and LAHF instructions deal with?

____

6.3 The SAHF/LAHF instructions will not let you modify which condition code?

____

6.2 Data Movement Instructions

Important data movement instructions include mov, xchg, lea, lds, les, lfs, lgs, lss, push, pop, pusha, popa, pushf, popf, lahf, and sahf. Keep in mind, *most* instructions move data around (often manipulating it at the same time). These instructions fall into this category because data movement is their primary goal[1].

6.2.1 MOV dest, source

The mov instruction copies data from a source operand to a destination operand. Chapter Four covers this instruction in detail, there is no need to consider it here.

The xchg instruction swaps the data between its operands. One operand must be a register, the other may be a register or a memory location. There is a special (one byte) version of this instruction that swaps AX with another register. The nop instruction is really a synonym for xchg ax, ax.

6.4 Suppose you want to exchange the values in the AX and BX registers. Using only MOV instructions and the AX, BX, and CX registers, write the code to do this:

____

____

____

6.5 Suppose you want to swap the 16-bit values in memory locations I & J. What three instructions will do this job (hint: only one is an XCHG):

____

____

____

1. ENTER and LEAVE are exceptions to this rule. We'll consider them here because they mesh so well with the PUSH/POP instructions.

6.2.2 LEA reg, memory

The lea (Load Effective Address) instruction computes the address of its memory operand and stores this address in the 16-bit register operand. This instruction also computes an effective address in addition to moving data around, so it is really an arithmetic statement, but most texts classify it as a data movement instruction.

The lea instruction is useful for setting up a pointer to some data structure in memory. For example, if you have a pointer to an array in bx and an index into that array in si, you can generate a pointer to the 3rd byte of that array element using the instruction lea bx, [bx+si+3]. From that point forward you can access the specified element by using the [bx] addressing mode. On many members of the 80x86 family, the simpler addressing mode is also faster (and it's always shorter). This is particularly useful if you can set up a register outside a loop and use a faster addressing mode inside the loop:

```
                  lea     bx, [bx+si+3]
ThisLoop:         mov     [bx], al
                  loop    ThisLoop
```

6.6 What will the "LEA AX, lbl[bx]" instruction do?

6.7 What is the difference between "LEA ax, [bx]" and "MOV ax, [bx]"?

6.8 Why isn't "LEA ax, bx" legal? Explain

The lds, les, lfs, lgs, and lss instructions let you load a 32-bit value into one of the 80x86's segment registers and a 16-bit general purpose register. The primary use of this instruction is to load a far pointer into a segment register and a pointer register. Although the instruction manipulates 32-bit quantities (even on eight and sixteen bit processors), you should not use this instruction for purposes other than initializing a segment/base register pair. Keep in mind that lfs, lgs, and lss are available only on 80386 and later processors.

As an example, consider the UCR Standard Library puts routine (see Chapter Six). This routine requires a pointer to a *zero terminated string* [2] in the es:di register pair. The following code will do this job:

```
String            byte    "string to print",0
StrPtr            dword   String
                   .
                   .
                   .
                  les     di, StrPtr
                  PUTS
```

2. A sequence of ASCII characters ending with a byte containing zero.

Note that the `les` instruction above *does not* copy the address of `StrPtr` into `es:di`.. Instead, it loads `es:di` with the 32 bits found at the source operand. The instruction `LES di, String` generates an error (see Chapter Six). Even if it would work, it would not load the address of `String` into `es:di`, it will load the four bytes "stri" into `es:di`.

6.9 What L*x*S instructions are available on all 80x86 processors?

____

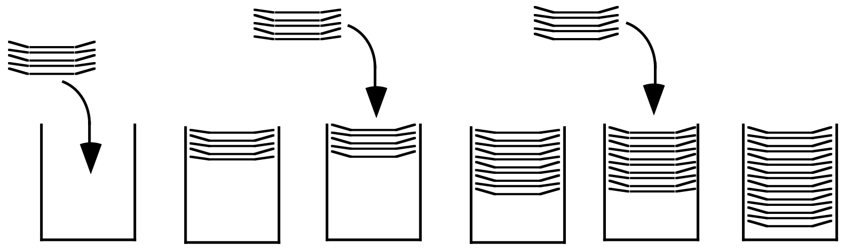
6.10 What will the LES DI, [BX+3] instruction do?

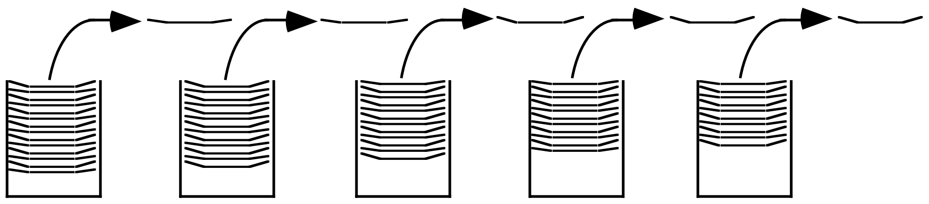
____

6.2.3 PUSH and POP

The `push` and `pop` instructions are the most complex in the data movement group. These instructions move data between registers/memory and the 80x86 *stack*. The stack is a region in memory contained within the *stack segment*, referenced by the 80x86 *stack pointer* register. The 80x86 CPUs automatically use the stack to store temporary data. You can also use the stack for temporary data using the `push` and pop instructions.

The 80x86 stack is a *last in first out* (LIFO) data structure. That is, the last thing you store onto the stack will be the first thing you retrieve from the stack. This type of data structure is particularly useful for various operations, particularly the subroutine call/return mechanism. The classic example of a stack is a dishwell at a local restaurant. The first plates the bus boy places into the dishwell are the *last* plates removed. Likewise, the last plates placed in the dishwell will be the first plates removed:


Pushing Plates onto the Stack


Popping Plates from the Stack

As you can see in the diagram above, the very first plate placed on the stack winds up on the bottom. It will be the very last plate removed from the dishwell. The 80x86 stack works in a similar fashion. The first values you push onto the stack are the last values you pop off the stack; the last values you push onto the stack are the first popped from the stack.

6.11 What does "LIFO" stand for?

____

6.12 If you were to push AX and BX on the stack (in that order), which value would you pop off the stack first?

The push instructions operate on 16-bit (push) or 32-bit (pushd) operands. You cannot push eight-bit values onto the 80x86 stack. The PUSH instructions subtract two from SP and then store the operand's value at memory location ss:sp. The pushd instructions (available only on 80386 and later processors) subtract four from sp and then store their operand at location ss:sp. Note that ss:sp always points at the last item pushed on the stack.

The pop and popd instructions operate in a similar fashion, undoing the push/pushd. They copy the value from memory location ss:sp to their destination operand, then add two or four to sp (two for pop, four for popd). Note that the 80x86 stack grows *down* in memory. That is, as you push things onto the stack, the stack uses lower and lower memory locations.

6.13 If SP contains 8010h and you push AX onto the stack, what will SP contain after the push?

6.14 If SP contains 7FFCh and you push EAX onto the stack, what will SP contain after the PUSHD?

6.15 If you pop the value on the stack into BX when SP contains 801Ch, what will SP contain after the pop?

The push/pushd instructions allow several different types of operands. You may push any of the 80x86's general purpose 16/32-bit registers, 16-bit segment registers, 16/32-bit memory locations, or 16/32-bit constants[3]. Except for constants, pop and popd support this same operand set.

The pushf and pushfd instructions push the flags register. The popf and popfd instructions pop the flags register. These instructions do not require any operands.

The 80286 and later processors also support "push all" and "pop all" instructions (pusha/pushad and popa/popad). These instructions push the set of general purpose registers onto the stack or pop them off the stack.

6.16 The 80x86 CPUs do not let you load immediate constants into a segment register. Describe how you could use a push and pop instruction on the 80286 and later processors to load a constant into DS:

6.17 If you push AX then BX onto the stack, and then pop AX and BX off the stack (in that order), you will exchange the values in AX and BX. Explain:

5.2 Carry, Aux Carry, Parity, Zero, and Sign.

5.3 Overflow

5.4 MOV CX, AX
MOV AX, BX
MOV BX, CX

5.5 MOV AX, I
XCHG AX, J
MOV I, AX

5.6 Loads the offset of "Lbl" plus the current value of BX into AX.

5.7 LEA AX, [BX] simply copies the value of BX into AX. MOV AX, [BX] loads the memory location pointed at by BX into AX

5.8 The second operand of LEA must be a memory addressing mode.

3. Pushing constants only works on 80286 and later processors, pushing 32-bit values only works on 80386 and later processors.

The PUSH and POP instructions are especially useful for temporarily saving the value in a register so you can use that register for some other purpose. For example, suppose you want to add word variables I and J together, storing the result into K, but all your general purpose registers contain important values. You could use the following code to resolve this dilemma:

```
push    ax
mov     ax, i
add     ax, j
mov     k, ax
pop     ax
```

There are a couple of very important rules you must always observe when using the stack to store temporary values. (1) You must always pop values off the stack in the *reverse* order that you push them onto the stack (see question 5.17 above). (2) If you push a value onto the stack, you must pop it off the stack. (3) Likewise, you shouldn't be popping values off the stack that you haven't pushed onto the stack.

A very common mistake in assembly language programs is to pop items from the stack in the same order they were pushed. Remember, the stack is a *last in first out* data structure. If you execute the instruction sequence:

```
push    ax
push    bx
push    cx
push    dx
```

followed by

```
pop     ax
pop     bx
pop     cx
pop     dx
```

the first pop instruction (pop ax) pops the *last* value pushed on the stack. This happens to be the value in DX at the time of the push sequence. Note that the sequence above effectively executes the instructions:

```
xchg    ax, dx
xchg    bx, cx
```

The push instructions always push the H.O. byte of their operand first. This guarantees that the pushed item appears in memory with it's L.O. byte at the lowest address (remember, the stack grows downward in memory). To match this, the pop instructions always pop the L.O. bytes first and the H.O. bytes last.

6.18 If you push EAX on the stack and then pop BX followed by AX, describe the values in the BX and AX registers after this sequence:

6.2.4 LAHF and SAHF

The `lahf` and `sahf` instructions copy the L.O. eight bits of the flags register into AH (`lahf`) or store AH into the L.O. eight bits of the flags register (`sahf`). Intel originally included these instructions in the instruction set for compatibility with their older 8080/8085 microprocessors. Today, no one cares about such compatibility so these instructions do not see much use. Their primary use is with the floating point instruction set on the 80x87 (see the chapter on floating point arithmetic) and as a means of directly manipulating many of the flags. One major drawback to these instructions is that the 80x86 overflow flag does not appear in the L.O. eight bits of the flags register.

6.3 Sign/Zero Extension and Conversion Instructions

The 80x86 family provides several instructions for performing various data conversions: bswap, cbw, cwd, cwde, cdq, movsx, movzx, and xlat.

5.9 LDS and LES

5.10 Load DI from location BX + 3 and load ES from location BX + 5.

6.3.1 BSWAP

Bswap (available only on 80486 and later processors) swaps bytes zero and three and bytes one and two in the destination register operand. This operation is useful for converting between 32-bit big and little endian values.

5.11 Last in first out

6.19 The BSWAP instruction only swaps the bytes in a 32-bit register. What 80x86 instruction can you use to swap the two halves of the AX register?

5.12 BX.

Suppose that you've obtain a binary image of an array of 32-bit integers created on a Motorola 68040 microprocessor (which uses big endian byte ordering). The following sequence of 80486 instructions will convert this array to the little endian data format the 80x86 processors use:

5.13 800Eh.

```
                  mov     cx, ArraySize          ;Number of elements .
                  lea     bx, Array              ;Ptr to 1st element.
Big2Little:       mov     eax, [bx]              ;Get next element.
                  bswap   eax                    ;Make little endian.
                  mov     [bx], eax              ;Save result away.
                  add     bx, 4                  ;Move no to next guy.
                  loop    Bit2Little
```

5.14 7FF8h.

6.3.2 CBW, CWD, CWDE, CDQ, MOVSX, and MOVZX

5.15 801Eh.

The cbw (convert byte to word) and cwd (convert word to double) instructions are available on all 80x86 processors. They sign extend al into ax (cbw) and ax into dx:ax (cwd). cwde also sign extends the 16-bit value in ax to 32 bits, but it leaves its result in eax rather than dx:ax. The Cdq instruction sign extends the 32-bit value in eax to a 64-bit value in edx:eax. The major drawback to these instructions is that they only operate on the accumulator. Furthermore, they leave their result in the accumulator and (possibly) the dx register. The movsx instruction eliminates these two restrictions by letting you move and sign extend any operands acceptable to the mov instruction. The only difference is that the destination operand must be larger than the source operand. The movzx instructions works in a similar fashion, but it zero extends rather than sign extends its operand.

5.16 PUSH constant
POP DS

6.20 What is an easy way to zero extend AL into AX using only a MOV instruction?

6.21 Given the solution to question 5.20, can you come up with an example which demonstrates why one would want to use the MOVZX instruction?

5.17 Since you pushed AX then BX, BX's value is the first popped off the stack. Hence if you execute POP AX followed by POP BX you will pop BX's old value into AX and AX's old value into BX.

Although these instructions are generally applicable anytime you want to extend the size of a signed integer value, it is especially crucial that you remember to zero or sign extend the accumulator operand when using the `div` and `idiv` instructions. Keep in mind that `div` and `idiv` divide the 16/32/64-bit values in `ax, dx:ax,` or `edx:eax` by an 8/16/32-bit operand. If you really want to perform an 8/8, 16/16, or 32/32 bit division, you can use the zero and sign extension instructions to extend the dividend to the appropriate size before the division takes place.

6.22 Suppose you want to divide the (signed) value in AX by the value in BX using the IDIV instruction. IDIV requires a 32-bit value in DX:AX prior to the division. Which instruction would you use to do this?

____

The following code sequence computes the average of the elements in an array. The array contains 16-bit signed integers and the number of elements in the array is given by the 16 bit variable *ArraySize*:

```
                mov     cx, ArraySize           ;# of items to ave.
                lea     bx, Array               ;Get ptr to array.
                mov     ax, 0                   ;Init sum.
AveLoop:        add     ax, [bx]                ;Add in next element.
                add     bx, 2                   ;Point at next item.
                loop    AveLoop

; AX now contains the sum of all items in the array.
; Compute the average by dividing by ArraySize.

                cwd                             ;Sign extend to dx:ax
                idiv    ArraySize

; Average is now in AX.
```

This code makes two important assumptions: `ArraySize` cannot be zero and the sum of all the array elements must fit into 16 bits.

6.3.3 XLAT

The `xlat` instruction performs a very simple, yet powerful, operation. It loads `al` with the value found in memory location `ds:[bx+al]`. Now this might seem like a somewhat esoteric operation, but it turns out to be quite useful. As its name implies, the `xlat` (translate) instruction lets you translate values from one form to another. For example, suppose you want to convert the L.O. nibble of AL into the hexadecimal character which represents that value (i.e., you want to convert 0..F to '0'..'F'). The following short section of code will do this:

```
HexTable        byte    "0123456789ABCDEF"
                 .
                 .
                 .
                and     al, 0Fh       ;Strip H.O. nibble
                lea     bx, HexTable  ;Load BX with Hextable's base address.
                xlat
```

This code assumes that the DS register points at the segment containing the HexTable array.

6.23 Suppose you have a table of 256 bytes which contain the values 0, 1, 2, 3, ... 0FFh. If BX points at the first byte of this table, what value will AL contain after the execution of the XLAT instruction? Explain

____

____

____

6.24 Suppose you have a table like the one above, except that it contains ‘a’ through ‘z’ at indices 41h through 5Ah (ASCII codes for ‘A’..’Z’). Explain what the XLAT instruction will do when used with this table.

5.18 BX will contain the original value in AX and AX will contain the original H.O. word of EAX.

5.19 XCHG AL, AH

6.25 Suppose you want to create a function which returns a one in AL if the character in AL is a punctuation character, a zero otherwise. Describe the table you would need to implement this function with the XLAT instruction:

5.20 MOV AH, 0

6.26 Suppose SI contains the input value for the function in problem 5.25 above. What instruction could you use to perform the same operation as the XLAT above?

5.21 MOVZX BX, AL or any other instruction whose destination is not and eight bit register or AX.

6.27 What could happen if you do not have exactly 256 bytes in the table pointed at by BX?

6.4 Arithmetic Instructions

‘The 80x86 arithmetic instructions include `add`, `adc`, `cmp`, `dec`, `div`, `idiv`, `imul`, `inc`, `mul`, `neg`, `sub`, and `sbb`. The are others, see the textbook for additional information.

The basic syntax these instructions is

```
add     dest, src
adc     dest, src
cmp     dest, src
```

```
                    dec     dest
                    div     src
                    idiv    src
                    imul    src
                    imul    dest, src, immediate
                    imul    dest, immediate
                    imul    dest, src
                    inc     dest
                    mul     src
                    neg     dest
                    sub     dest, src
                    sbb     dest, src
```

Only the first version of the IMUL instruction above is available on all versions of the 80x86 family. The remaining versions require an 80286 or later processor (see the textbook). The other instructions are all available on every member of the 80x86 family.

6.4.1 ADD, ADC, SUB, SBB, and CMP

The add/adc, sub/sbb, and cmp instructions work for both signed and unsigned operands. Given the nature of the two's complement system, they perform *both* operations simultaneously. The syntax for these instruction is

```
                    add     dest, source
                    adc     dest, source
                    sub     dest, source
                    sbb     dest, source
                    cmp     dest, source
```

The add instruction computes dest := dest + source and sets the condition code flags depending on the result. Likewise, the sub instruction compute dest := dest - source and sets the flags accordingly. The adc and sbb instructions work just like the add and sub instructions except they add in or subtract out the value of the carry flag (zero or one) in addition to the source operand. This lets you take the overflow from one computation and add it into another.

6.28 If the carry flag is set, what will the instruction "ADC AX, BX" do?

____

6.29 What will the above instruction do if the carry flag is clear?

____

Since the SHL instruction copies the H.O. bit of its operand into the carry flag and the ADC instruction can add the current value of the carry flag into an operand, it is very easy to construct a short loop which will count the number of one bits in an operand. Consider the following loop that provides one way to count the number of set bits in the bx register:

```
                    mov     cx, 16              ;Do 16 bits.
                    mov     ax, 0               ;Initialize sum to zero.
CountBits:          shl     bx, 1               ;SHR would work, too.
                    adc     ax, 0               ;Add this bit to the sum.
                    loop    CountBits
```

The cmp instruction works like the SUB instruction except it does not store the difference into the destination operand. Instead, it simply updates the 80x86's flags register. You can use the conditional set and jump instructions after a cmp instruction to test the result of the comparison.

6.30 The CMP is a *non-destructive* operation because it does not modify the value of either operand. Give an example of an instruction which performs the same operation as CMP but is a *destructive* operation:

____

The cmp instruction will set the carry flag if the first (unsigned) operand is less than the second; that is, the jb and jc instructions are one and the same. The following example takes advantage of this fact. It determines whether a given value is greater than or less than the median in an array of values. It does this by counting the number of entries in the array which are smaller than the value in dx, multiplies this result by two, and then compares this to the number of elements in the array:

```
                  lea     bx, Array             ;Init ptr to data.
                  mov     cx, ArraySize         ;# of items.
                  mov     dx, TestValue         ;Check this value.
                  mov     ax, 0                 ;Init sum.
MedianSum:        cmp     dx, [bx]              ;C=1 if less than.
                  adc     ax, 0                 ;Bump less than cntr.
                  add     bx, 2                 ;On to next element.
                  loop    MedianSum

; AX now contains the number of array elements which
; were less than the value of TestValue. If this value,
; multiplied by two, is greater than the number of
; elements in the array then TestValue is greater than
; the median value. Otherwise it is less than or
; equal to the median value.

                  shl     ax, 1                 ;Multiply by two.
                  cmp     ax, ArraySize         ;See if > median
                  ja      AboveMedian           ;Go elsewhere if >.

; If the code falls through to this point, then TestValue
; is less than or equal to the median value.
```

5.22 CWD

5.23 It will not affect AL at all since you will be loading AL with the value already present in AL.

5.24 It will convert upper case characters to lower case since if AL contains 'A' through 'Z' prior to XLAT it will load 'a' through 'z' from the corresponding locations in the table. All other values are unchanged.

5.25 Put a zero into each entry of the table whose index is *not* the ASCII code of a punctuation character. Put a one in the entries corresponding to punctuation chars.

6.4.2 INC and DEC

The inc and dec instructions add one or subtract one from their single operand. They behave just like the add dest,1 and sub dest,1 except that these instructions do not affect the carry flag.

6.31 Besides the fact that the INC and DEC instructions do not affect the carry flag, what is a good reason to use these instructions rather than ADD or SUB?

____

____

____

5.26
MOV AL, [BX+SI]

5.27 If the value in AL is greater than the maximum number of bytes in the table the XLAT instruction will fetch a byte from beyond the table.

6.4.3 NEG

The neg instruction takes the two's complement of its operand, thereby negating it. Note that this is the same thing as subtracting the destination operand from zero or (following the standard definition of two's complement) inverting all the bits and adding one. The neg instructions sets the flags in a manner identical to the sub instruction assuming the destination operand was zero before the subtraction. Hence, the flags will be set as though you had compared zero to the neg operand.

6.32 Under what circumstance(s) will the NEG instruction set the zero flag?

____

____

Although the 80x86 does not have an instruction that computes the absolute value of an integer operand, building one using the `neg` instruction is very easy. The following two code segments demonstrate how to take the absolute value of the `ax` register:

```
; Straight-forward example: If AX is negative, negate
; it. This method works best if the number is usually
; negative since the code will not have to branch as
; often.

                  cmp    ax, 0
                  jns    NotNeg
                  neg    ax
NotNeg:

; Second example. This one works best if the number is
; usually positive:

                  neg    ax
                  jns    NotPos
                  neg    ax
NotPos:
```

6.4.4 MUL, IMUL, DIV, and IDIV

The multiplication and division instructions come in two basic flavors: `mul/div` and `imul/idiv`. These instructions handle unsigned operations (`mul/div`) and signed operations (`imul/idiv`). Although the two's complement system performs signed and unsigned addition/subtraction in an identical manner, signed and unsigned multiplication and division are different operations requiring different instructions.

The `mul` instruction allows only a single memory or general purpose register operand. It is an extended precision unsigned multiply operation that multiplies its operand with the accumulator (`al/ax/eax`) and produces a result that is twice the size of the accumulator (the result goes into `ax`, `dx:ax`, or `edx:eax`, depending on the operand size). The `mul` instruction sets the carry and overflow flags if the result does not fit in the same number of bits as the source operand. Note, however, that the result is still correct since `mul` produces a result twice as large as the source operand[4].

The `imul` instruction performs signed arithmetic. This instruction takes several forms:

• IMUL operand	;extended acc := acc * operand
• IMUL reg, mem/reg, constant	;reg = mem/reg * constant
• IMUL reg, constant	;reg = reg * constant
• IMUL reg, reg/mem	;reg = reg * mem/reg

The single operand `imul` instruction works just like the `mul` instruction except it works with signed values. The remaining of `imul` instructions (those with two or three operands) compute a result that is the same size as the source operands. Since the multiplication result may not fit in the destination operand, the result may not be exact. In the event of overflow, `imul` sets the carry and overflow flags and throws away the high order bits of the result.

6.33 What will "`IMUL AX, BX, 10`**" compute?**

____

4. The product of two n-bit numbers will always fit in no more than 2*n bits.

The versions of the imul instruction that allow immediate operands are especially useful for computing offsets into a multi-dimensional array. For example, consider the following Pascal array:

```
UseMul:array [0..5,0..74] of char;
```

To store a zero into “UseMul[i,j]” you could use the 80286 code:

```
imul    bx, i, 75
add     bx, j
mov     UseMul[bx], 0
```

This version of the imul instruction is available only on 80286 and later processors, but it is very convenient if you are using one of these processors.

6.34 Suppose you have the array “ThreeD:array [0..4, 0..5, 0..6] of char;” What instructions could you use to load element “ThreeD[i,j,k]” into AL

____

____

The following short code segment produces the *cross product* of two arrays. The cross product is the sum of the products of corresponding array elements, e.g.,

```
sum := 0;
for i := 0 to ArraySize do
       sum := sum + A[i] * B[i];
```

The assembly language code that computes this is

```
                  mov    cx, ArraySize         ;# of elements.
                  mov    bx, 0                 ;Init array index.
                  mov    bp, 0                 ;Hold result here.
CrossLp:          mov    ax, A[bx]             ;Get A[i].
                  imul   B[bx]                 ;Multiply by B[i].
                  add    bp, ax                ;Sum up products.
                  add    bx, 2                 ;On to next element.
                  loop   CrossLp

; BP now contains the cross product.
```

Note that this code ignores the possibility of overflow. There are actually two places where overflow can occur. First, the imul instruction may produce a value in dx which is not 0 or 0FFFFh. Second, the add bp, ax instruction could overflow and this code does not check for that.

The div and idiv instructions require a single operand, like the extended precision versions of mul and imul . There are no special 80286/386 versions of these instructions like imul. Div is for unsigned operations, idiv is for signed operations. These instructions compute the quotient and remainder at the same time. They divide a 64-bit value by a 32-bit value, a 32-bit value by a 16-bit value, or a 16-bit value by an eight-bit value. The operand is the smaller value, the implied operand is either edx:eax, dx:ax, or ax. These instructions place the quotient in the lower half of the implied operand and the remainder in the upper half of the implied operand. *If the quotient does not fit into eax, ax, or al (as appropriate), the div/idiv instructions generate a divide error.*

6.35 Suppose you have the value 20000h in DX:AX and you divide this by the value in BX. What three values in BX will cause the divide instruction to generate a divide error?

____

5.28 Compute AX := AX + BX + 1

5.29 Compute AX := AX + BX + 0

5.30 SUB.

5.31 They are shorter and generally faster than the ADD equivalent.

6.36 Besides not being able to fit the quotient in the destination register, what other error can the divide instruction encounter?

____

Since division by zero will immediately bomb your program, you should always check your divisor when using the div and idiv instructions if you are not absolutely sure they are non-zero. The following code demonstrates how to check for this when computing K:=I div J:

```
cmp     J, 0            ;Divisor zero?
je      DivideBy0       ;Error handler code.
mov     ax, I
mov     dx, 0           ;Remember to zero extend!
div     J
mov     K, ax
```

Handling the second major problem with division, producing a quotient which will not fit into the destination register, is a little more difficult. The solution most people adopt is to perform an n-bit by n-bit division rather than a 2n-bit by n-bit division. The example above demonstrates this. I, J, and K are all 16-bit variables. This code simply zero extended ax into dx before doing the division (presumably the variables are all unsigned).

6.5 Logical, Shift, Rotate, and Bit Instructions

This group of important instructions includes and, or, xor, not, ror, rol, rcr, rcl, shl/sal, shr, sar, shrd, shld, bt, bts, btr, btc, bsf, bsr, and set*cc*. Their syntax is

```
AND     dest, source
OR      dest, source
XOR     dest, source
NOT     dest
SHL     dest, count     ;SAL is a synonym for SHL.
SHR     dest, count
SAR     dest, count
SHLD    dest, source, count
SHRD    dest, source, count
RCL     dest, count
RCR     dest, count
ROL     dest, count
ROL     dest, count
TEST    dest, source
BT      source, index
BTC     source, index
BTR     source, index
BTS     source, index
BSF     dest, source
BSR     dest, source
SETcc   dest                   ;See textbook for cc.
```

The and, or, xor, and not instructions perform bitwise logical operations on their operands allowing you to clear selected bits, set selected bits, invert selected bits, or invert all the bits in their destination operands.

If you only consider the value of a single bit in their operands, then you can treat these operations like the boolean operators in Pascal or C.However, a common mistake is to “mix metaphors” when using these instructions. For example, C programmers often treat zero as false and anything else as true. In particular, most programmers use one for true and zero for false. One must be careful, however, because “not 1” does not equal false, it equals 0FEh which is true! Likewise, 0AAh (true) logically ANDed with 55h (true) is equal to zero (false).

6.37 One way to convert these instructions to “boolean logical instructions” is to use only bit zero for true and false and reset all other bits to zero after each operation. Which of the log-

ical operations can you use to force all bits except bit zero to the value zero? What operand would you use with this logical operation to accomplish this?

5.32 If you negate zero it produces zero and sets the zero flag.

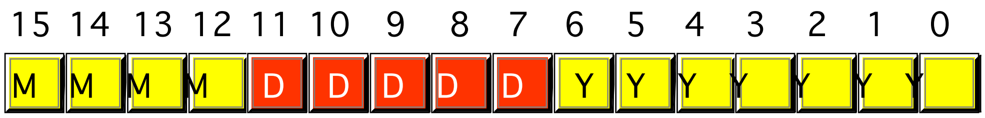
6.38 Given that A, B, C, D, and E are all Pascal boolean variables (bytes), what 80x86 code could you use to implement the following Pascal statement:

A := (B and C) or not (D and E);

5.33 AX := BX * 10

Despite their similarity to Pascal boolean operators, the and, or, xor, and not instructions are quite useful for decidedly non-boolean operations. In particular, you can use these instructions to set, clear, and invert arbitrary bits in an register or memory location. Recall the DATE data type from Chapter Two

5.34
imul bx, i, 6
add bx, j
imul bx, bx, 7
add bx, k
mov al, ThreeD[bx]



Suppose ax contains a value in the range 1..31 which you wish to insert into the day field of this date structure. The first thing you will need to do is shift the days value seven positions to the left to align it with "DDDDD" field above. There are a couple of ways to do this. On 80286 and later processors the best way is probably to use the shl reg, 7 instruction. Once you have shifted the value into place, you will need to replace the existing DDDDD field with the new data. Probably the easiest way to do this is to force the existing bits to zero and then logically OR in the new DDDDD data:

5.35 Zero, one, or two.

```
; Insert Day value in AX into the DDDDD field of the date structure
; in the BX register

                  shl    ax, 7          ;Properly position day bits.
                  and    bx, 0F07Fh     ;Clear existing DDDDD bits.
                  or     bx, ax         ;Merge in the new day value.
```

The following code extracts the year and returns the current year if it is a leap year, otherwise it returns the previous leap year in ax. Note that this code does *not* work for the years 1900..1903 since 1900 was not a leap year. It does, however, work for 2000..2003.

```
; Return last leap year. Assume DATE value is in the AX
; register upon entering this section of code.

                  and    ax, 7ch
```

Yes, that's not a misprint. It only took one machine instruction. To bad the explanation of how it works is considerably longer!

ANDing the date value with 7Fh, of course, strips out the year field and sets all other bits to zero. To get the previous leap year (or the current year if it is a leap year) all we've got to do is compute "Year - (Year MOD 4)". If the current year is evenly divisible by four, "Year MOD 4" is zero and the calculation above returns the current year. If last year was a leap year, then

"Year MOD 4" is one, which subtracted from this year's date produces a leap year. Likewise for the other two possibilities.

ANDing a value with 0FCh produces the same value. "AX MOD 4" always produces a value between zero and three, which fits nicely into two bits. If you force the L.O. two bits of ax to zero you effectively reset its value back to the last multiple of four, which is exactly what we want.

The and AX, 7ch combines the two AND operations into one: it masks out everything except the year field and it clears the two L.O. bits of the year field in a single operation.

One interesting use of the XOR operation is to exchange data between two operands. If you XOR anything with itself, you get zero. The following sequence of operations will swap the values A and B without using a temporary memory location:

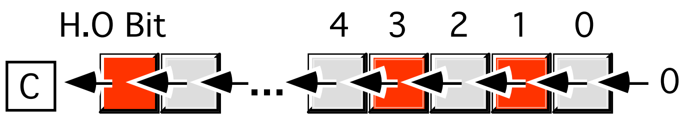
```
A := A xor B;        B := A XOR B;        A := A XOR B;
```

Of course, on the 80x86 you can use the xchg instruction to swap two operands, however, suppose you want to swap the MONTH field in our data type with the value in the ax register. The following code will do this without using any extra registers. Try it!

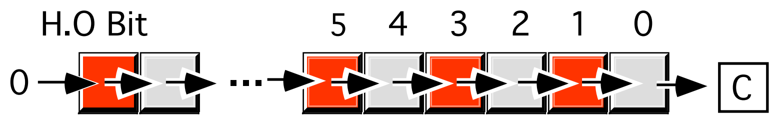
```
ror     ax, 4           ;Assume month was 1..12.
xor     ax, Date
xor     Date, ax
xor     ax, Date
rol     ax, 4           ;Put month into range 1..12.
```

6.5.1 SHL, SHR, SAR, SHLD, SHRD, RCL, RCR, ROL, and ROR

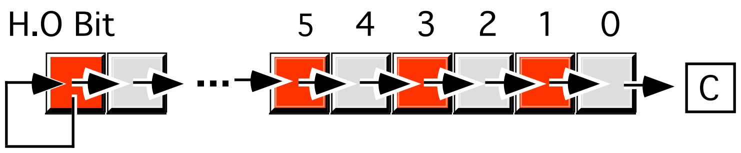
The 80x86 shift instructions perform the following operations:



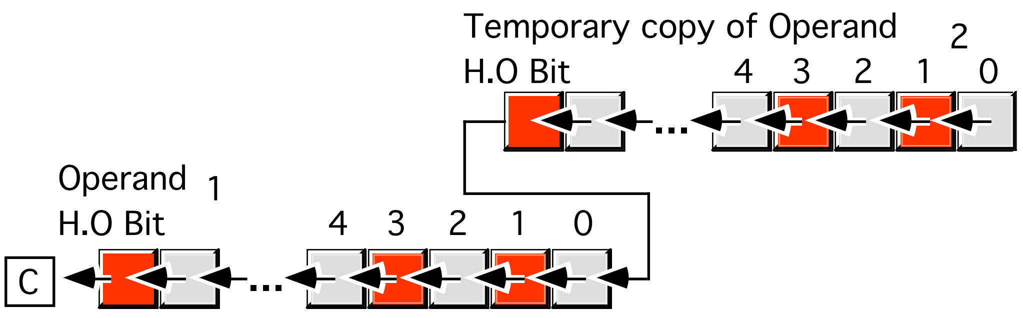
Shift Left Operation



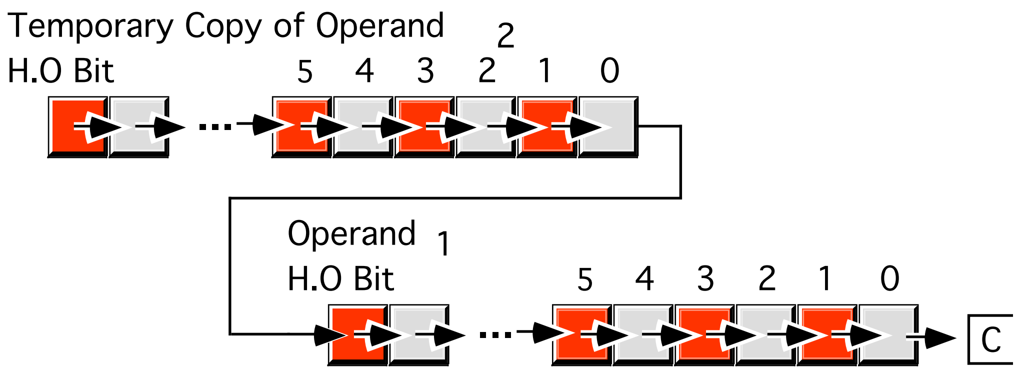
Shift Right Operation



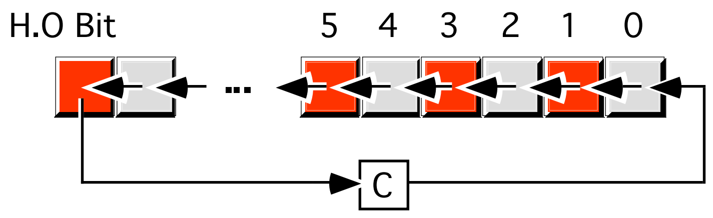
Arithmetic Shift Right Operation



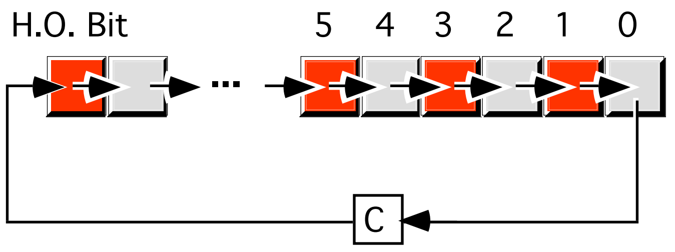
Double Precision Shift Left Operation



Double Precision Shift Right Operation



Rotate Through Carry Left Operation

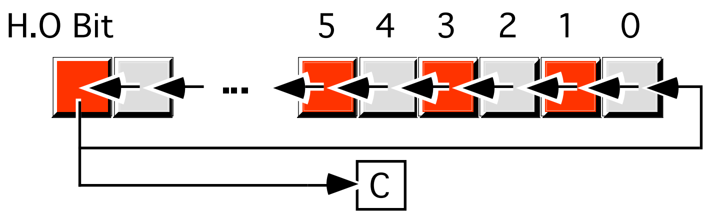


Rotate Through Carry Right Operation

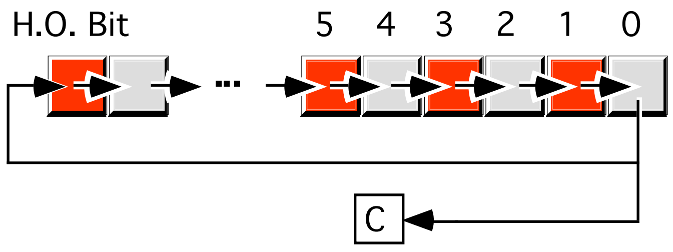
5.36 Division by zero.

5.37 AND, 1

5.38
mov al, b
and al, c
mov ah, d
and ah, e
not ah
and ah, 1
or al, ah
and al, 1
mov a, al



Rotate Left Operation



Rotate Right Operation

The syntax for these instructions is

```
shl     reg/mem, 1
shl     reg/mem, cl
shl     reg/mem, integer            (2)

sal     reg/mem, 1
sal     reg/mem, cl
sal     reg/mem, integer            (2)

shr     reg/mem, 1
shr     reg/mem, cl
shr     reg/mem, integer            (2)

sar     reg/mem, 1
sar     reg/mem, cl
sar     reg/mem, integer            (2)

shld    reg/mem, reg, integer       (3)
shld    reg/mem, reg, cl            (3)

shrd    reg/mem, reg, integer       (3)
shrd    reg/mem, reg, cl            (3)

rcl     reg/mem, 1
rcl     reg/mem, cl
rcl     reg/mem, integer            (2)

rcr     reg/mem, 1
rcr     reg/mem, cl
rcr     reg/mem, integer            (2)

rol     reg/mem, 1
rol     reg/mem, cl
rol     reg/mem, integer            (2)

ror     reg/mem, 1
ror     reg/mem, cl
ror     reg/mem, integer            (2)
```

```
(2) Available only on 80286 and later processors.
(3) Available only on 80386 and later processors.
```

The `shl` and `sal` mnemonics are synonyms. Note that `shl` by one bit corresponds to a multiplication by two. `Shr` and `sar` deal with unsigned and signed values, respectively. When shifting one bit to the right, they correspond to a division by two.

6.39 What instruction(s) could you use to multiply the value in AX by four? (Hint: don't use ADD, MUL or IMUL)

____

6.40 When dividing an integer value by two, it's quite possible that you will end up with an inexact result (when dividing an odd number by two). How can you test after SAR/SHR to see if you've obtained an inexact result?

____

____

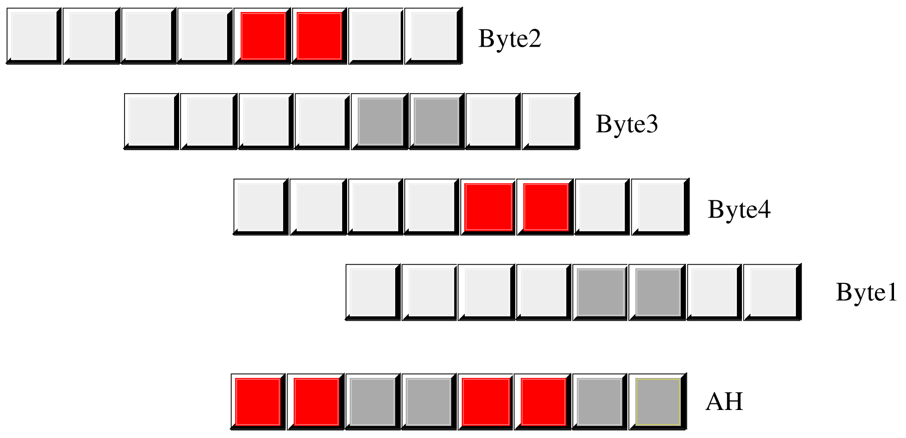
6.41 How can you use the AND instruction to convert AX to zero if it is even, one if it is odd?

____

The rotate instructions come in handy when you need to pack several values into a single byte, word, or double word. For example, suppose you have four bytes and merge bits two and three from each of these four bytes into a single byte. You could use the following code to accomplish this:

```
mov     ah, Byte1
and     ah, 00001100b          ;Mask out our bits.
rol     ah, 2                  ;Move them over!
mov     al, Byte2
and     al, 00001100b          ;Mask out other bits.
or      ah, al                 ;Merge into value.
rol     ah, 2
mov     al, Byte3
and     al, 00001100b
or      ah, al
rol     ah, 2
mov     al, Byte4
and     al, 00001100b
or      ah, al
```

The code above shifts Byte1, Byte2, Byte3, and Byte4 into ah as follows:



6.5.2 TEST

The `test` instruction logically ANDs its two operands but does not store the result in the destination operand. It simply sets the flags according to the result of the logical AND operation.

```
test   dest, source
```

6.42 What other instruction works in a non-destructive fashion like TEST? What is the operation it performs?

____

6.43 Explain how you use the TEST instruction to see if the value in AX is even or odd. Why is this method better than using the AND instruction?

____

____

____

You can use the `test` instruction to check the value of an individual bit within some value. If one of the operands contains a single set bit, then `test` will set the zero flag if the corresponding bit in the other operand is zero; it will clear the zero flag if that corresponding bit contains a one. Therefore, you can use the `test` instruction to see if an individual bit is set or clear.

Another common use of the `test` instruction is to see if a group of bits in some value are all zero. If one of the operands contains one bits in all the positions you require zeros in the second operand, the CPU will set the zero flag only if all the corresponding bits are zero. For example, `and al, 0fh` sets the zero flag only if bits zero through three of `al` contain zeros.

6.5.3 BT, BTS, BTR, and BTC

The Bit Test (`bt`), Bit Test and Set (`bts`), Bit Test and Reset (`btr`), and Bit Test and Complement (`btc`) instructions also let you test individual bits. The second operand of these instructions specifies a bit number (0..31) in the first operand. They copy that bit into the carry flag so you can test the carry flag afterwards. The `bts`, `btr`, and `btc` instructions also set, clear, or complement the bit after copying it into the carry flag. These instructions are available only on 80386 and later processors. Their syntax is

```
bt      reg/mem, reg
bt      reg/mem, integer

bts     reg/mem, reg
bts     reg/mem, integer

btr     reg/mem, reg
btr     reg/mem, integer

btr     reg/mem, reg
btr     reg/mem, integer
```

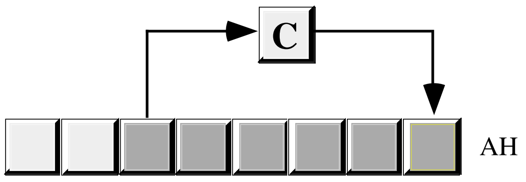
The `bt` instruction provides a convenient way to test an individual bit within some value. This is usually more straight forward than using the `test` instruction since you can specify the exact bit you want to test rather than a binary mask. The `bts`, `btr`, and `btc` instructions let you easily manipulate *flags* (boolean variables). For example, suppose you want to test a boolean variable to see if it is false and set it to true regardless of its current setting. You can easily accomplish this with a single `bts` instruction:

```
bts     flag, 0              ;Assume value is in bit zero.
```

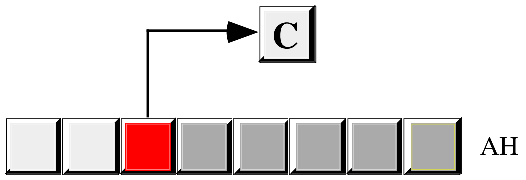
One use for the `btr` instruction is to perform an n-bit `rol` operation. Consider the following code which rotates six bits to the right in AL:

```
btr     al, 5           ;Bit #5->carry, set to zero.
rcl     al, 1           ;Copy carry to bit zero, etc.
bt      al, 0           ;Set carry to original value.
```

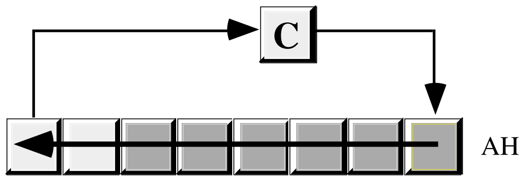
The `btr` instruction copies bit five into the carry flag and then clears bit five. The `rcl` instruction rotates the data in `al` one position to the left and copies the carry flag into bit zero. The bt instruction copies the rotated bit back into the carry flag, producing the desired result:



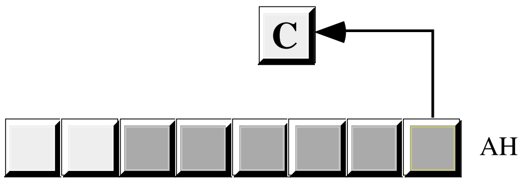
Six Bit ROL Operation



BTR copies bit five to the carry and clears bit five



RCL copies the carry into bit zero and shifts the data to the left one position (note that bit five contains zero before the shift)



BT copies bit zero back into the carry flag

6.44 Explain how you could use the BT and ADC instructions to round all odd numbers to the next higher even value:

____

____

5.39 SHL AX, 2 (on 80286 or later) or the sequence:

SHL AX, 1
SHL AX, 1

5.40 Look at the carry flag. If it's set the result is inexact.

5.41 AND AX, 1

6.6 BSF and BSR

The `bsf` and `bsr` instructions look for the first set bit in an operand starting from bit zero (`bsf`) or the H.O. bit (`bsr`). They return (in the destination operand) the bit count of the position of the first set bit in the source operand:

```
bsf    reg, reg/mem
bsr    reg, reg/mem
```

These instructions scan through the reg/mem operand looking for the first set bit. They store the bit position of that bit in the reg operand. If the reg/mem operand is not zero (i.e., there is at least one set bit), then these instruction clear the zero flag. If the reg/mem operand is zero, these instructions set the zero flag and the reg operand contains an indeterminate result.

6.45 BSF and BSR only look for the first set bit in a value. If you want to search for the first zero bit in a value, you must perform some operation on the data to convert all the zeros to ones and vice versa. What is this operation?

____

6.46 Suppose you want to clear the set bit detected by "BSR AX, Bits" immediately after executing the BSR instruction. What single instruction could you use to do this that uses the value in AX?

____

6.6.1 SETcc

The `set`*cc* (set on condition) instructions test one or two bits in the flags register and then store a byte containing zero or one in the destination operand depending on the status of the flag bit(s). This instruction is quite useful for synthesizing complex boolean operations. `set`*cc* tests the following conditions: `setc` (carry set), `setnc` (carry clear), `setz` (zero flag set), `setnz` (zero flag clear), `sets` (sign flag set), `setns` (sign flag clear), `seto` (overflow flag set), `setno` (overflow flag clear), `setp` (parity flag set), `setnp` (parity flag clear), `seta` (unsigned greater than), `setae` (unsigned greater or equal), `setb` (unsigned less than), `setbe` (unsigned less than or equal), `sete` (equal), `setne` (not equal), `setg` (signed greater than), `setge` (signed greater or equal), `setl` (signed less than), and `setle` (signed less than or equal). There are some aliases for these instructions as well, see the textbook for details. These instructions are available only on 80386 and later processors.

You can use the `set`*cc* instructions to easily perform complex boolean computations. Consider the following Pascal statement:

```
B := ((i = j) and (k <= l)) or (m <> n);
```

This Pascal code easily converts to the following 80386 code using the `set`*cc* instructions:

```
mov    ax, i        ;BL := i = j
cmp    ax, j
sete   bl
mov    al, k        ;BH := k <= l
cmp    ax, l
setle  bh
and    bl, bh       ;BL := BL and BH
mov    ax, m        ;BH := m <> n
cmp    ax, n
setne  bh
or     bl, bh       ;BL := BL or BH
mov    b, bl        ;b := bh
```

Since set*cc* always produces zero or one, you can use the and, or, and xor instructions to perform the corresponding boolean operations. Keep in mind, however, that if you want to use the not operation to negate a boolean value, you will need to follow it up with an and to remove the unnecessary bits.

5.42 CMP, SUB

6.47 To perform the boolean not operation on a multi-bit value, you really only want to invert the L.O. bit, not all the bits like the NOT operation. What instruction can you use to invert only the L.O. bit?

____

6.7 The I/O Instructions

5.43 TEST AL, 1 (or TEST AX, 1) sets the zero flag if the number is even. This is better than AND because it is non destructive.

The 80x86 provides four I/O instructions: in, out, ins, and outs. The in and out instructions read data from the I/O address space (in) and write data to the I/O address space (out). We'll consider ins and outs in the next section.

There are two forms of the in and out instructions:

```
in      al, port
in      al, dx
out     port, al
out     dx, al
```

The port operand is an eight-bit constant in the range 0..255. The 80x86 CPU family supports up to 65,536 different I/O locations. The instructions utilizing the port address operand can only address the first 256 locations in this 16-bit I/O address space. To access the other locations you must load the address into the dx register and utilize the in/out instructions with the dx operand.

The IBM PC printer port typically appears at address 378h in the I/O address map. You can output data to the LPT port by writing a byte to this I/O address. The following 80x86 code will store the character 'A' on the printer port data lines:

```
mov     dx, 378h
mov     al, 'A'
out     dx, al
```

5.44 Assuming the value is in AX:
BT AX, 0
ADC AX, 0

6.48 You can read the last byte written to the parallel port using the IN instruction. What sequence of instructions would you use to read the 'A' written above from the parallel port?

____

____

Please note that the above instructions are not sufficient if you actually want to print a character to the printer on the parallel port. The in and out instructions simply set up the data for the printer to print. Additional instructions (which we'll discuss in a later chapter) are necessary to actually print data to the printer.

6.8 The String Instructions

The 80x86 string instructions manipulate blocks of bytes in memory. Indeed, they are the only instructions which allow you to directly move data from one memory location to another or compare one memory location to another without going through any registers. However, because of the setup involved when using these instructions, you would not normally use them to manipulate just a few bytes. They are best utilized to manipulate large blocks of data.

The string instructions perform the following operations:

```
MOVS{b,w,d}:      ES:[DI] := DS:[SI]
                  if direction_flag = 0 then
                         SI := SI + size;
                         DI := DI + size;
                  else
                         SI := SI - size;
                         DI := DI - size;
                  endif;

LODS{b,w,d}:      EAX/AX/AL := DS:[SI]
                  if direction_flag = 0 then
                         SI := SI + size;
                  else
                         SI := SI - size;
                  endif;


STOS{b,w,d}:      ES:[DI] := EAX/AX/AL
                  if direction_flag = 0 then
                         DI := DI + size;
                  else
                         DI := DI - size;
                  endif;

CMPS{b,w,d}:      CMP DS:[SI], ES:[DI]         ;Set appropriate flags.
                  if direction_flag = 0 then
                         SI := SI + size;
                         DI := DI + size;
                  else
                         SI := SI - size;
                         DI := DI - size;
                  endif;

SCAS{b,w,d}:      CMP    EAX/AX/AL, ES:[DI]    ;Set appropriate flags.
                  if direction_flag = 0 then
                         DI := DI + size;
                  else
                         DI := DI - size;
                  endif;

INS{b,w,d}:       ES:[DI] := port(DX)
                  if direction_flag = 0 then
                         DI := DI + size;
                  else
                         DI := DI - size;
                  endif;

OUTS{b,w,d}:      port(DX) := DS:[SI]
                  if direction_flag = 0 then
                         SI := SI + size;
                  else
                         SI := SI - size;
                  endif;
```

size = 1, 2, or 4 for bytes, words, or double words.

Although the term "string instructions" conjures up some magic operations you can use to manipulate character strings, there is nothing about these instructions that limits them to manipulating character strings. Indeed, you'll use these instructions for other data types as often as you use them to manipulate character strings.

Consider the movs instruction. Successively executing this instruction copies data from one array to another. For example, if you have an array of 16 words, the following code will copy data from one array to another:

```
Ary1Ptr          dword  Array1
Ary2Ptr          dword  Array2
                   .
                   .
                   .
                 lds    si, Ary1Ptr
                 les    di, Ary2Ptr
                 mov    cx, 16
CopyArray:       movsw
                 loop   CopyArray
```

Using a string instruction in a loop as above is so common that Intel created a special *prefix* instruction which merges the loop instruction above into the movs instruction. If you place the rep (repeat) prefix just before the movsw above, you do not need the loop instruction:

```
                 lds    si, Ary1Ptr
                 les    di, Ary2Ptr
                 mov    cx, 16
          rep    movsw
```

The lods and stos instructions, executed together, perform the same operation as the movs instruction [5]. That is, the following loop does the same work as the rep movsb instruction:

```
DoMOVS:          lodsb
                 stosb
                 loop   DoMOVS
```

Since the lodsb/stosb method is bigger and slower, you might be wondering why anyone would want to use this sequence. Well, the main reason is because you can insert additional instructions between the lodsb and stosb instructions. For example, you could place the xlat instruction between them to perform some data translation while copying one array of bytes to another (e.g., convert all the characters in one string to uppercase while moving them).

```
                 mov    cx, SizeOfString
ConvertLoop:     lodsb
                 cmp    al, 'A'         ;convert upper case to
                 jb     NotAlpha        ; lower case letters.
                 cmp    al, 'Z'
                 ja     NotAlpha
                 and    al, 0Bfh        ;Clear bit 6 (see chapter one).
NotAlpha:        stosb
                 loop   ConvertLoop
```

6.49 Although it is legal to use the REP prefix with the LODS instruction, doing so doesn't make much sense. Why? (Hint: what happens if you execute two LODSB instructions in a row?)

____

____

5. Except, of course, the LODS/STOS instruction pair modify the AL/AX/EAX register while MOVS does not.

5.45 NOT

5.46 BTR Bits, AX

5.47 XOR. Assuming the value is in AX you could use the instruction
XOR AX, 1
to invert only the L.O. bit.

Another possibility is to use the instruction:
btc ax, 0

5.48
mov dx, 378h
in al, dx

Though you wouldn't ever want to use the REP prefix with the lods instruction, it makes perfect sense to use it with the stos instruction. You can use rep to force stos to fill an array with a single value. For example, the following code zeros out the 16-element integer array pointed at by Ary1Ptr:

```
                les     di, Ary1Ptr
                mov     cx, 16
                mov     ax, 0
        rep     stosw
```

You'll get a chance to see how the other string instructions work with the REP/REPE/REPNE prefixes in the chapter on string instructions.

6.9 Unconditional Jumps

The unconditional jump instructions transfer control to the instruction specified by the operand of the jump. Unlike most HLLs, the 80x86 jmp instruction provides many different types of jump operations. The reason is quite simple: most HLLs want to discourage the programmer from using gotos, especially fancy ones. However, the compilers themselves need to generate fancy goto/jmp operations in order to convert high-level structures (like case/switch, while, etc.) into machine code.

80x86 jmp instructions come in two basic categories: near (intrasegment) and far (intersegment). The near jmp instructions can transfer control to any statement within the current code segment. These instructions simply copy their operand into the ip register. They allow the following syntactical variations:

```
                jmp     disp
                jmp     reg
                jmp     mem16
```

6.50 In many respects, the JMP instruction is really just another form of the MOV instruction. Explain:

____

____

Near jmp operands can be an immediate value, a 16-bit general purpose register, or a 16-bit memory location (addressed by any of the 80x86's addressing modes). If an immediate operand is present, MASM converts it to an eight-bit or 16-bit signed displacement that provides the distance from the jmp instruction to the target location. If you specify a register operand, the jmp instruction simply copies the data from the specified register into ip. If you specify a memory location, using a memory addressing mode, the jmp instruction copies the 16 bits from the specified memory location into ip.

6.51 What is the difference between the "JMP BX" and "JMP [BX]" instructions? Explain what each of these do:

____

____

____

____

The far jmp copies its operand into the cs:ip register pair. It behaves like an "lcs ip,*operand*" instruction If the operand is an immediate value, it provides the full segment:offset address of the target location; this is not a displacement to the target. If the operand of a far jmp is a memory location, then it is a double word value providing the segment (in the H.O. word) and offset (in the L.O. word) of the target location. There is no far jmp instruction that uses register operands.

Most jmp instructions are near jumps with an immediate operand. The chapter on control structures in the textbook goes into the gory details of how you would use the other forms of the jmp instruction. In general, it's best if you avoid the indirect jumps (those using memory or register operands) since they can make your programs much harder to read and maintain.

5.49 Doing so would load each new value over the top of the old value in AL/AX/EAX. Hence you would lose the results of all but the last operation.

6.10 The CALL and RET Instructions

The CALL and RET instructions provide the mechanism for implementing procedures, functions, and other types of program units.

The call instruction pushes a *return address* onto the stack and then jumps to the address given by its operand. Other than the fact that call pushes the address of the instruction following the call onto the stack, it behaves exactly like the jmp instruction.

One minor complication is that the near and far call instructions push different values on the stack. The near call pushes a 16-bit offset into the current code segment onto the stack before doing a near jump. the far call pushes a 32-bit return address, cs followed by the offset, onto the stack.

The ret instruction pops the return address off the stack and transfers control to that address. There are two types of return instructions: near and far. The near return instruction simply pops a 16-bit value off the stack into the ip register. The far return instruction pops a 16-bit value into ip and a second 16-bit value into cs. There are two other forms of the return instruction, we will consider their operation in Chapter Nine.

6.11 The INT, INTO, BOUND, and IRET Instructions

The into, bound, and iret instructions are advanced instructions which require an understanding of the 80x86 interrupt structure prior to using them. For that reason, we will not consider these instructions again until the chapter on interrupts and concurrency. In general, the same could be said for the int instruction except that the IBM PC system uses the int instruction to transfer control to DOS and other system related routines. One does not need to know how the int instruction operates to use it to make calls to MS-DOS or the PC's ROM BIOS routines.

The ROM BIOS (Basic Input/Output System) on the IBM PC provides a routine you can use to test and read data from the keyboard. Although the exact entry point in the ROM is not guaranteed, you can call the routine wherever it sits in memory using the int 16h instruction. Loading ah with zero prior to executing int 16h instructs the BIOS routine to read a key from the keyboard (and not return from INT 16h until you press a key) and return the ASCII code for that key in al. Likewise, BIOS uses the int 10h instruction with ah=0Eh to print the character in al to the video display. So the following loop reads characters from the keyboard until the user presses the Enter key (ASCII code is 0dh) and displays those characters on the video display:

```
ReadLoop:       mov     ah, 0
                int     16h             ;Read a key from keyboard.
                mov     ah, 0eh
                int     10h             ;Write char to display.
                cmp     al, 0dh         ;See if ENTER key.
                jne     ReadLoop        ;Repeat if not ENTER.
```

For more information about the interrupt routines that DOS and BIOS support, see the chapter on BIOS and DOS in your textbook.

6.12 The Conditional Jump Instructions

The conditional jump instructions are some of the most important instructions in the 80x86 instruction set. These instructions let your programs make decisions. These Jcc instructions (there is a corresponding j*cc* instruction to every Set*cc* instruction) test one or two bits in the flags register and transfer control to a target location if the condition is true. If the condition is not true, control falls through to the next instruction after the conditional jump.

Valid Jcc instructions include jc, jnc, jz, jnz, js, jns, jo, jno, jp, jnp, ja, jae, jb, jbe, je, jne, jg, jge, jl, and jle (there are some other aliases as well, see your textbook). These instructions test the same conditions as the set*cc* instructions. See that section for more details.

The j*cc* instructions are particularly useful after the cmp instruction to check the result of the that operation. Together, the cmp and j*cc* instructions let you synthesize IF/THEN/ELSE instructions and loops like WHILE or REPEAT/UNTIL. For example, the following code adds 5 to ax if ax is less than bx:

```
                cmp     ax, bx
                jnb     SkipAdd
                add     ax, 5
SkipAdd:
```

6.52 Write a short code segment which swaps the values in AX and BX if AX is less than BX:

____

____

____

____

The 8086-80286 j*cc* instructions will jump a maximum of ±128 bytes around the current instruction. If the branch is out of range the easiest resolution is to convert it to a pair of jumps, the opposite j*cc* instruction jumping around a jmp to the actual target location. Generally, MASM versions 6.0 and later will automatically convert an out of range j*cc* for you. Once in a while it will fail on you, however. Furthermore, you may need to perform this conversion manually yourself if you are not using an assembler which converts out of range jumps. The following code demonstrates how to translate an out of range jc instruction to one which will work fine:

Original Code:

```
                shl     ax, 1
                jc      OutOfRange
```

Converted Code:

```
                shl     ax, 1
                jnc     SkipMe
                jmp     OutOfRange
SkipMe:
```

6.53 Suppose that the instruction "JE RepeatLoop" is out of range. What is the code that will correct this problem?

____

____

____

In general, most programmers use je/jne after a cmp instruction and they use jz/jnz after most other instructions which affect the zero flag. Of course, you could use je/jz or jne/jnz interchangeably but good programming style mandates that you use these instructions where appropriate. For example, the cmp and sub instructions set the zero flag in exactly the same way. However, it makes sense to talk about two operands being *equal* after a cmp or a subtraction producing a *zero* result. Hence you'll normally use je/jne after cmp and jz/jnz after sub. Here are some examples:

```
cmp     ax, 1               ;See if AX = 1.
je      AXisOne

test    al, 1               ;See if bit zero is one.
jnz     Bit0Is1

test    ax, ax              ;See if AX = 0.
jz      AXisZero

inc     ax                  ;See if AX overflows
jz      Overflow
```

Jc and jnc , and their synonyms jb and jnb are also very common in assembly language programs. Since many instructions affect the carry flag (e.g., arithmetic, shift, and rotate instructions) you'll often use jc and jnc . Like je/jnz you will typically use jb/jnb after a compare instruction.

```
cmp     al, 'A'             ;See if AL < 'A'
jb      NotAlpha

shr     ax, 1               ;Move L.O. bit into carry
jc      LOBitIsSet          ; and branch if set.
shr     ax, 1               ;Now move bit #1
jc      Bit1IsSet           ; into carry and branch
etc.                        ; if set.
```

The 80x86 conditional branch instructions are quite useful for synthesizing loops and IF..THEN..ELSE statements. But that is a subject for another chapter.

5.50 It moves its operand into the IP register.

5.51 JMP BX jumps to the memory location whose address is in BX. The JMP word ptr [bx] jumps to the address specified by the word at memory location DS:[bx].

6.13 The JCXZ and LOOPxx Instructions

The 80x86 provides four instructions which manipulate and test the CX register and then conditionally branch to some target location: jcxz/jecxz, loop, loope, and loopne.

Jcxz checks the cx (ecx for jecxz) register to see if it is zero. If so, jcxz transfers control to the target address, if cx is not zero, control transfers to the next instruction after jcxz. Jcxz is particularly useful in conjunction with the loop instruction (described next) to handle the case when cx is zero.

The loop instruction decrements cx and then branches to the target location if cx is not zero. *Note that the test for CX=0 occurs after the decrement operation.* If cx is zero when you first execute the loop instruction, loop decrements cx (producing 0FFFFh) which is not zero so control transfers to the specified target location. E.g., the following loop will execute 65,536 times:

```
; Presumably CX contains zero at this point...

LoopToHere:         mov     [bx], ax
                    add     bx, 7
                    loop    LoopToHere
```

6.54 Fix the above loop so that it will not execute 65,536 times if CX turns out to be zero upon entering the loop:

___

___

___

___

___

The loop and loopne instructions also decrement cx and fall through if cx is zero, but they also test the zero flag to determine if the branch should be taken. The following code reads up to 80 characters unless a carriage return comes along before then:

```
                  mov           cx, 80
ReadTillCR:       mov           ah, 0                 ;Read a character
                  int           16h                   ; from the keyboard
                  <do something with AL>
                  cmp           al, 0dh               ;See if return
                  loopne        ReadTillCR
```

The printer port *busy* flag is bit seven of input port 379h on many systems. A *printer spooler* program under MS-DOS will often check for some period of time to see if the printer is busy. If it remains busy for an extended period of time DOS will continue other processing. However, if it clears the busy bit after a period, the printer spooler will send another character to the printer. The following code demonstrates how to do this with the LOOPNE instruction:

```
TestBusy:         mov    cx, 4000h      ;Time to wait for busy.
                  mov    dx, 379h       ;Printer status port.
BusyLoop:         in     al, dx         ;Get printer port status.
                  test   al, 80h        ;Test the busy bit.
                  loopne BusyLoop       ;Repeat while busy
                  jne    StillBusy
          <Output another char here>
                  jmp    TestBusy       ;Try again
StillBusy:
```

6.14 Miscellaneous Instructions

The 80x86 provides several additional instructions you can use to manipulate various flags and otherwise control the operation of the CPU. These instructions include clc (clear carry) stc (set carry), cmc (complement carry), cld (clear direction flag), std (set direction flag), cli (clear interrupt flag), sti (set interrupt flag), nop (no operation), and hlt.

The nop instruction is really a synonym for the xchg ax,ax instruction. This is a one byte instruction which doesn't affect any registers or flags and consumes one to three clock cycles (depending on the processor). The main purpose for this instruction is to "punch out" instructions in memory while using a debugger like CodeView. If you discover that you've got an instruction you don't want in the object code, you can replace each byte of the instruction with a nop byte (opcode = 90h) and the program will continue to function assuming it doesn't have any real tight timing dependencies.

The hlt instruction stops the CPU dead in its tracks. After executing hlt , only hardware interrupts and the reset line are operational. For a typical PC system, if you execute hlt you will need to reboot the machine.

6.15 Using MASM and LINK

In the laboratory exercises for this chapter you will need to use MASM and LINK to assemble the output produced by the IBM/L (Instruction Bench Marking Language) system. This section describes some of the steps you will need to follow to use MASM with the IBM/L system.

Assuming no special file dependencies, using MASM 6.x is very easy. Just type

```
ml name.asm
```

and you're in business. MASM will assemble and link your program producing an ".EXE" executable file.

In the laboratory you will linking the output of the IBM/L compiler with some routines in the UCR Standard Library. If you've installed the UCR Standard Library routines on your hard disk (the Standard Library code appears on the disk accompanying this manual) then you need to execute the following DOS commands to properly set MASM's *include* and *lib* paths:

```
SET INCLUDE=C:\STDLIB\INCLUDE
SET LIB=C:\STDLIB\LIB
```

Of course, if you've placed the standard library "include" and "lib" files in directories other than those listed above, you will need to make the appropriate changes to these DOS commands. Furthermore, if you use another Microsoft language or some other system that needs include and lib paths set up, you may need to adjust the above lines for those languages as well. Please see the manual accompanying those other languages.

Another possibility is to copy all the include and lib files from the UCR Standard Library into your working directory. However, this tends to clutter your working directory. You should only use this approach if you cannot modify the include or lib path for one reason or another.

5.52
```
        cmp ax, bx
        jae NoSwap
        xchg ax, bx
NoSwap:
```

6.16 IBM/L (Instruction Benchmarking Language)

IBM/L lets you time sequences of instructions to see how much time they *really* take to execute. The cycle timings in most 80x86 assembly language books are horribly inaccurate as they assume the absolute best case. IBM/L lets you try out some instruction sequences and see how much time they actually take. This is an invaluable tool to use when optimizing a program. You can try several different instruction sequences that produce the same result and see which sequence executes fastest.

IBM/L uses the system 1/18th second clock and measures most executions in terms of clock ticks. Therefore, it would be totally useless for measuring the speed of a single instruction (since all instructions execute in *much* less than 1/18th second). IBM/L works by repeatedly executing a code sequence thousands (or millions) of times and measuring that amount of time. IBM/L automatically subtracts away the loop overhead time.

IBM/L is a compiler which translates a source language into an assembly language program. Assembling and running the resulting program benchmarks the instructions specified in the IBM/L source code and produces relative timings for different instruction sequences. An IBM/L source program consists of some short assembly language sequences and some control statements which describe how to measure the performance of the assembly sequences. An IBM/L program takes the following form:

5.53
```
        je Skip
        jmp RepeatLoop
Skip:
```

```
#data
           <variable declarations>
#enddata

#unravel <integer constant>
#repetitions <integer constant>
#code ("title")
%init
           <initial instructions whose time does not count>
%eachloop
           <Instructions repeated once on each loop, ignoring time>
%discount
           <instructions done for each sequence, ignoring time>
%do
           <statements to time>
#endcode

<Additional #code..#endcode sections>

#end
```

Note: the %init, %eachloop, and %discount sections are optional.

IBM/L programs begin with an optional data section. The data section begins with a line containing "#DATA" and ends with a line containing "#ENDDATA". All lines between these two lines are copied to an output assembly language program inside the dseg data segment. Typically you would put global variables into the program at this point.

Example of a data section:

```
#DATA
I                 word   ?
J                 word   ?
K                 dword  ?
ch                byte   ?
ch2               byte   ?
#ENDDATA
```

These lines would be copied to a data segment the program IBM/L creates. These names are available to *all* #code..#endcode sequences you place in the program.

Following the data section are one or more code sections. A code section consists of optional #repetition and #unravel statements followed by the actual #code..#endcode sections.

The #repetition statement takes the following form:

```
           #repetition integer_constant
```

(The "#" must be in column one). The integer constant is a 32-bit value, so you can specify values in the range zero through two billion. Typical values are generally less than a few hundred thousand, even less on slower machines. The larger this number is, the more accurate the timing will be; however, larger repetition values also cause the program IBM/L generates to run much slower.

This statement instructs IBM/L to generate a loop which repeats the following code segment *integer_constant* times. If you do not specify any repetitions at all, the default is 327,680. Once you set a repetitions value, that value remains in effect for all following code sequences until you explicitly change it again. The #repetition statement must appear outside the #code..#endcode sequence and affects the #code section(s) following the #repetition statement.

If you are interested in the straight-line execution times for some instruction(s), placing those instructions in a tight loop may dramatically affect IBM/L's accuracy. Don't forget, executing a control transfer instruction (necessary for a loop) flushes the pre-fetch queue and has a big effect on execution times. The #unravel statement lets you copy a block of code several times inside the timing loop, thereby reducing the overhead of the conditional jump and other loop control instructions. The #unravel statement takes the following form:

```
           #unravel count
```

(The "#" must be in column one). *Count* is a 16-bit integer constant that specifies the number of times IBM/L copies the code inside the repetition loop.

Note that the specified code sequence in the #code section will actually execute (*count * integer_constant*) times, since the #unravel statement repeats the code sequence count times inside the loop.

In its most basic form, the #code section looks like the following:

```
#CODE ("Title")
%DO
        <assembly statements>
#ENDCODE
```

The title can be any string you choose. IBM/L will display this title when printing the timing results for this code section. IBM/L will take the specified assembly statements and output them inside a loop (multiple times if the #unravel statement is present). At run time the assembly language source file will time this code and print a time, in clock ticks, for one execution of this sequence.

Example:

```
#unravel 16                     16 copies of code inside the loop
#repetitions 960000             Do this 960,000 times
#code ("MOV AX, 0 Instruction")
%do
                  mov    ax, 0
#endcode
```

The above code would generate an assembly language program which executes the mov ax,0 instruction 16 * 960000 times and report the amount of time that it would take.

Most IBM/L programs have multiple code sections. New code sections can immediately follow the previous ones, e.g.,

```
#unravel 16                     16 copies of code inside loop
#repetitions 960000             Do the following code 960000 times
#code ("MOV AX, 0 Instruction")
%do
                  mov    ax, 0
#endcode

#code ("XOR AX, AX Instruction")
%do
                  xor    ax, ax
 #ENDCODE
```

The above sequence would execute the mov ax, 0 and xor ax, ax instructions 16*960000 times and report the amount of time necessary to execute these instructions. By comparing the results you can determine which instruction sequence is fastest.

Any statement that begins with a semicolon in column one is a comment which IBM/L ignores. It does not write this comment to the assembly language output file.

All IBM/L programs must end with a #end statement. Therefore, the correct form of the program above is

```
#unravel 16
#repetitions 960000
#code ("MOV AX, 0 Instruction")
%do
                  mov    ax, 0
#endcode
#code ("XOR AX, AX Instruction")
%do
                  xor    ax, ax
#ENDCODE
#END
```

5.54
```
        jcxz skiploop
LoopToHere:
        mov [bx], ax
        add bx, 7
        loop LoopToHere
skiploop:
```

An example of a complete IBM/L program using all of the techniques we've seen so far is

```
#data
                  even
i                 word   ?
                  byte   ?
j                 word   ?
#enddata

#unravel 16
#repetitions 32, 30000
#code ("Aligned Word MOV")
%do
                  mov    ax, i
#endcode

#code ("Unaligned word MOV")
%do
                  mov    ax, j
#ENDCODE
#END
```

There are a couple of optional sections which may appear between the #code and the %do statements. The first of these is %init which begins an initialization section. IBM/L emits initialization sections before the loop, executes this code only once. It does not count their execution time when timing the loop. This lets you set up important values prior to running a test which do not count towards the timing. E.g.,

```
#data
i                 dword  ?
#enddata
#repetitions 100000
#unravel 1
#code
%init
                  mov    word ptr i, 0
                  mov    word ptr i+2, 0
%do
                  mov    cx, 200
lbl:              inc    word ptr i
                  jnz    NotZero
                  inc    word ptr i+2
 NotZero:         loop   lbl
#endcode
#end
```

Sometimes you may want to use the #repetitions statement to repeat a section of code several times. However, there may be some statements that you only want to execute once on each loop (that is, without copying the code several times in the loop). The %eachloop section allows this. Note that IBM/L does not count the time consumed by the code executed in the %eachloop section in the final timing.

Example:

```
#data
i                 word   ?
j                 word   ?
#enddata

#repetitions 40000
#unravel 128
#code
%init -- The following is executed only once

                  mov    i, 0
                  mov    j, 0
```

```
%eachloop -- The following is executed 40000 times, not 128*40000 times

                  inc    j

%do -- The following is executed 128 * 40000 times

                  inc    i

#endcode
#end
```

In the above code, IBM/L only counts the time required to increment i. It does not time the instructions in the %init or %eachloop sections.

The code in the %eachloop section only executes once per loop iteration. Even if you use the #unravel statement (the inc i instruction above, for example, executes 128 times per loop iteration because of #unravel). Sometimes you may want some sequence of instructions to execute like those in the %do section, but not count their time. The %discount section allows for this. Here is the full form of an IBM/L source file:

```
#DATA
           <data declarations>
#ENDDATA
#REPETITIONS value1, value2
#UNRAVEL count
#CODE
%INIT
           <Initialization code, executed only once>
%EACHLOOP
           <Loop initialization code, executed once on each pass>
%DISCOUNT
           <Untimed statements, executed once per repetition>
%DO
           <The statements you want to time>
#ENDCODE
<additional code sections>
#END
```

To use this package you need several files. IBML.EXE is the executable program. You run it as follows:

```
c:> IBML filename.IBM
```

This reads an IBML source file (filename.IBM, above) and writes an assembly language program to the standard output. Normally you would use I/O redirection to capture this program as follows:

```
c:> IBML filename.IBM >filename.ASM
```

Once you create the assembly language source file, you can assemble and run it. The resulting EXE file will display the timing results.

To properly run the IBML program, you must have the IBMLINC.A file in the current working directory. This is a skeleton assembly language source file into which IBM/L inserts your assembly source code. Feel free to modify this file as you see fit. Keep in mind, however, that IBM/L expects certain markers in the file (currently ";##") where it will insert the code. Be careful how you deal with these existing markers if you modify the IBMLINC.A file.

The output assembly language source file assumes the presence of the UCR Standard Library for 80x86 Assembly Language Programmers. In particular, it needs the STDLIB include files (stdlib.a) and the library file (stdlib.lib).

In Chapter One of this lab manual you should have learned how to set up the Standard Library files on your hard disk. These must be present in the current directory (or in your

INCLUDE/LIB environment paths) or MASM will not be able to properly assemble the output assembly language file. For more information on the UCR Standard Library, see the next chapter.

The following are some IBM/L source files to give you a flavor of the language.

```
; IBML Sample program: TESTMUL.IBM.
; This code compares the execution
; time of the MUL instruction vs.
; various shift and add equivalents.

#repetitions 480000
#unravel 1

; The following check checks to see how
; long it takes to multiply two values
; using the IMUL instruction.

#code ("Multiply by 15 using IMUL")
%do
                  .286
                  mov    cx, 128
                  mov    bx, 15
MulLoop1:         mov    ax, cx
                  imul   bx
                  loop   MulLoop1

#endcode

; Do the same test using the extended IMUL
; instruction on 80286 and later processors.

#code ("Multiplying by 15 using IMUL")
%do
                  mov    cx, 128
MulLoop2:         mov    ax, cx
                  imul   ax, 15
                  loop   MulLoop2

#endcode

; Now multiply by 15 using a shift by four
; bits and a subtract.

#code ("Multiplying by 15 using shifts and sub")
%init
%do
                  mov    cx, 128
MulLoop3:         mov    ax, cx
                  mov    bx, ax
                  shl    ax, 4
                  sub    ax, bx
                  loop   MulLoop3

#endcode
#end
```

Output from TESTMUL.IBM:

```
              IBM/L 2.0

Public Domain Instruction Benchmarking Language
 by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to ±2.

CPU: 80486

Computing Overhead: Multiply by 15 using IMUL
Testing: Multiply by 15 using IMUL
Multiply by 15 using IMUL :370
```

```
Computing Overhead: Multiplying by 15 using IMUL
Testing: Multiplying by 15 using IMUL
Multiplying by 15 using IMUL :370
Computing Overhead: Multiplying by 15 using shifts and sub
Testing: Multiplying by 15 using shifts and sub
Multiplying by 15 using shifts and sub :201
```

```
; IBML Sample program MOVs.
; A comparison of register-register
; moves with register-memory moves

#data
i                 word   ?
j                 word   ?
k                 word   ?
l                 word   ?
#enddata

#repetitions 30720000
#unravel 1

; The following check checks to see how
; long it takes to multiply two values
; using the IMUL instruction.

#code ("Register-Register moves, no Hazards")
%do
                  mov    bx, ax
                  mov    cx, ax
                  mov    dx, ax
                  mov    si, ax
                  mov    di, ax
                  mov    bp, ax
#endcode

#code ("Register-Register moves, with Hazards")
%do
                  mov    bx, ax
                  mov    cx, bx
                  mov    dx, cx
                  mov    si, dx
                  mov    di, si
                  mov    bp, di
#endcode

#code ("Memory-Register moves, no Hazards")
%do
                  mov    ax, i
                  mov    bx, j
                  mov    cx, k
                  mov    dx, l
                  mov    ax, i
                  mov    bx, j
#endcode

#code ("Register-Memory moves, no Hazards")
%do
                  mov    i, ax
                  mov    j, bx
                  mov    k, cx
                  mov    l, dx
                  mov    i, ax
                  mov    j, bx
#endcode
#end
```

```
          IBM/L 2.0

Public Domain Instruction Benchmarking Language
 by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to Ò 2.

CPU: 80486

Computing Overhead: Register-Register moves, no Hazards
Testing: Register-Register moves, no Hazards
Register-Register moves, no Hazards :25
Computing Overhead: Register-Register moves, with Hazards
Testing: Register-Register moves, with Hazards
Register-Register moves, with Hazards :51
Computing Overhead: Memory-Register moves, no Hazards
Testing: Memory-Register moves, no Hazards
Memory-Register moves, no Hazards :67
Computing Overhead: Register-Memory moves, no Hazards
Testing: Register-Memory moves, no Hazards
Register-Memory moves, no Hazards :387
```

6.17 The CPUDYNO Program

The CPUDYNO program is another program you can use to test the timings of various instructions on your computer. The main purpose of this program is to compute cycle timings for individual instructions and various addressing modes on the CPU you're using. Although your text book publishes Intel/Microsoft instruction timings in the appendices, the published times are often *best case* timings that are difficult to achieve in real programs. Running CPUDYNO allows you to time a set of 80x86 instructions under less than ideal circumstances to get a more realistic instruction execution time.

The timing differences between the Intel published timings and the results CPUDYNO reports are generally due to wait states, prefetch queue loading, cache effects, hazards, instruction prefix bytes, and other timing retardants mentioned only in the fine print in Intel's manuals.

To use CPUDYNO, simply run "CPUDYNO" from the DOS command line. It will display the timings for several 8086 instructions on the screen. You can use I/O redirection to save the output for comparison purposes.

6.18 The 80x86 Instruction Set Laboratory Exercises

In this laboratory you will study instruction encodings and timings for several common 8086 instructions. You will use MASM, CodeView, and IBM/L to prepare and test short assembly language programs. You will also learn how to use the linker to combine various object modules and how to set up pathnames for the UCR Standard Library routines. Finally, you will run the CPUDYNO program to test the execution time of various instructions (in cycles) on your computer system (and several different systems, if available).

6.18.1 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide chapter with all the questions answered and corrected.
- A write-up on the IBM/L language explaining, in your own words, how the program works.
- A description of what the CPUDYNO program does.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

6.18.2 Laboratory Exercises

In this laboratory you will perform the following activities:

- Assemble some instructions inside CodeView and inspect their binary encoding.
- Assemble some short assembly language programs using ML and load the result into CodeView.
- Observe how several instructions affect the 80x86 flags register.
- Generate several instruction sequences and compare their relative timings using the IBM/L language.
- Run the CPUDYNO program and compare the timings it reports against the official Intel timings.

❏ Exercise 1: Assemble and link the "SHELL.ASM" file (supplied on the diskette accompanying this lab manual). Be sure that the standard library include and lib files are in the MASM include and lib paths. Assemble SHELL.ASM using the following command:

```
ml /Zi /Fl shell.asm
```

The "/Zi" option tells ML to specially prepare this file for use with CodeView. It instructs ML to include special information in the .EXE file so CodeView can provide *symbolic source code* debugging facilities. You should use this option whenever you plan to use CodeView to debug your program. You should *not*, however, use this option all the time because it makes the ".EXE" file larger.

The "/Fl" command line item is optional. It instructs ML to create an *assembly listing* of the source file. This listing file includes your source code plus other information including the binary translation of each instruction in the program. (displayed in hexadecimal).

Once you've assembled the file (without error), note that ML created *three* new files (assuming they were not present already): SHELL.OBJ, SHELL.EXE, and SHELL.LST. The SHELL.LST file is the one containing the assembly listing. Load this file into your editor or print it out to the printer to see the format of an ML listing file.

For your lab report: include the SHELL.LST listing file.

For additional credit: Execute the DOS command "ML /?" and have MASM list out all the legal command line options. Explain the purpose of several of these command line options.

❏ Exercise 2: Load the "LAB5A.ASM" file into the editor and locate the comment which states "Put your main program here." At this point, insert the following 80x86 assembly language statements:

```
add     ax, bx
sub     ax, bx
and     ax, bx
or      ax, bx
xor     ax, bx

add     bx, ax
add     bx, ds:[0]
add     bx, [bx]
add     bx, 200h[bx]

xor     ax, ax
xor     ax, 1
xor     ax, 100h

xchg    ax, ax
nop

mov     al, 12h
mov     ax, 1234h
```

Assemble this code using the ML command:

```
ml /Zi /Fl lab5a.asm
```

(Note: if the assembler reports an error in the above statements, correct the error in the source file and reissue the ML command.)

For your lab report: Edit or print the LAB5A.LST file and look at the binary encodings for each of these instructions. Explain each of the particular values produced by the assembler.

For additional credit: look up the instruction encodings for each of these instructions in Appendix D of your textbook. Describe the meaning of each of the hexadecimal values the assembler emits to the object code file.

❏ Exercise 3: Load the LAB5A.EXE file created by ML above into CodeView using the DOS command:

```
CV LAB5A
```

By default, CodeView will display the file in *source* mode. What you will see in the source window is the text from the LAB5A.ASM file. While this is probably the best mode to use when actually debugging a program (since you see the comments and symbolic names), it will actually interfere with this exercise. To switch from “source” to “assembly” mode, select the “Source Window” item from the “Options” menu. This brings up a dialog box. One of the options in this dialog box is “Display Mode.” Select “Assembly” as the display mode option. You will note that the source display window changes from a nice source display to a disassembly listing. This is fine, the source display hides a lot of important information we want to take a look at.

The first thing you should do is locate the instruction sequence you entered in Exercise 2 above. After finding this sequence, compare the disassembled code against your assembly listing. Report and explain any differences between the assembly listing and the CodeView disassembly.

Note: one difference you will find is the way MASM displays 16-bit (and 32-bit) values in the assembly listing. Normally, MASM displays bytes of object code emitted by the assembler from left to right with the leftmost byte corresponding to the value at the lowest address. For example, MASM typically displays the instruction “MOV AL, 12h” as follows:

```
BO 12              MOV    AL, 12h
```

When MASM encounters a 16 (or 32) bit operand you would expect that it would display the object code bytes from left to right corresponding to L.O. to H.O. However, for such constants MASM swaps the operand bytes so that they read more naturally:

```
B8 1234            MOV    AX, 1234h
```

rather than:

```
B8 34 12           MOV    AX, 1234
```

Note that MASM does *not* put a space between the values when it reverses the byte order. This is how you can tell that it is displaying the bytes in the reverse order. CodeView does not reverse these bytes when it disassembles your code in “assembly” mode.

For your lab report: List or otherwise emphasize all instructions in the program that use this display form for 16 bit values.

❏ Exercise 4: While in CodeView, assemble the following instruction sequences into memory starting at location 8000:0 (using the command window assemble command).

```
mov     al, 0ffh
add     al, 1

mov     al, 7fh
add     al, 1

mov     al, 80h
add     al, 1

mov     al, 0ffh
inc     al

mov     al, 80h
inc     al

int     3
```

Single step through each of the instructions above and note the results in the AL register and the carry, sign, overflow, and zero flags after the execution of each instruction. Comment on the results. (Do not execute the “int 3” instruction. It's present in case you accidentally execute the "GO" command. It will stop program execution inside CodeView.)

For your lab report: Describe the condition code settings (carry, overflow, zero, and sign) after the execution of each instruction and discuss why the 80x86 produces each flag setting.

❏ Exercise 5: Repeat exercise 4 for the following sequence of compare instructions. Be sure to describe what the flag values mean with respect to the signed and unsigned comparisons being performed.

```
                mov     al, 0
                cmp     al, 0
                cmp     al, 1
                cmp     al, 0ffh              ;-1 or +255

                mov     al, 1
                cmp     al, 0
                cmp     al, 1
                cmp     al, 2
                cmp     al, 0ffh              ;-1 or +255

                mov     al, 0ffh
                cmp     al, 0feh              ;-2/+254
                cmp     al, 0ffh
                cmp     al, 0
                cmp     al, 1
```

For your lab report: Discuss the condition code values after the execution of each of the above CMP instructions.

For additional credit: Try some other "boundary" values to when comparing values. Repeat the exercise above.

❏ Exercise 6: Create an IBM/L program which times the following code sequences. Execute these code sequences to determine which ones are the fastest. If you have access to several different CPUs (e.g., 80286, 80386, 80486) try running the IBM/L output on each of them.. Note: if you are running a machine which emulates the 80x86 using a program such as SoftPC, the timing values you obtain will be meaningless. IBM/L timings are only meaningful when run on a true 80x86 CPU.

For all sequences, place two word variables I and J in the data section. Initialize these variables to five and ten, respectively.

```
; Sequence 1a:

#data
i                 word   ?
j                 word   ?
#enddata
#unravel 256
#repetitions 250000
#code ("Add one with ADD")
%do
                  mov    ax, i
                  add    ax, 1
                  mov    i, ax
#endcode
#end

Sequence 1b:

#data
i                 word   ?
j                 word   ?
#enddata
#unravel 256
#repetitions 250000
#code ("Add one with INC")
%do
```

```
                  inc     i
#endcode
#end

Sequence 2a:

#data
i                 word    ?
j                 word    ?
#enddata
#unravel 256
#repetitions 250000
#code ("*4 with MUL")
%do
                  mov     al, 4
                  mul     i
#endcode
#end

Sequence 2b:

#data
i                 word    ?
j                 word    ?
#enddata
#unravel 256
#repetitions 250000
#code ("*4 with SHL")
%do
                  mov     ax, i
                  shl     ax, 1
                  shl     ax, 1
#endcode
#end
```

For your lab report: compile the code sequences above using IBM/L and include the results in your lab report.

For additional credit: Merge the four sequences above into a single IBM/L program. Create several additional sets of sequences on your own and run these through IBM/L. Use IBM/L to demonstrate the cost of hazards on an 80486 or later processor. Compile and run the sample IBM/L files on the diskette accompanying this lab manual. Explain the results.

❏ Exercise 7: The IBM/L program is particularly useful to compare the timings of two or more different code fragments that all achieve the same result. Consider the absolute value operation mentioned earlier in this chapter. There were two possible sequences, one that works best for positive values, one that works best for negative values. There are several other ways to implement the absolute value function. The following code sequence achieves this without using any control transfer instructions:

```
                  cwd                   ;DX becomes 0FFFFh if AX is negative.
                  xor     ax, dx        ;Invert AX if it is negative.
                  and     dx, 1         ;Convert DX to one (if FFFF) or zero.
                  add     ax, dx        ;Add one if AX was negative.
```

The only question is, "which sequence is better?" One way to answer this question is to write a short IBM/L program and try these three segments out. The following code does just that:

```
#repetitions 2000000
#unravel 100

#code      ("ABS" Sequence 1 w/positive value")
%discount
                  mov     ax, 1         ;Our positive value
%do
                  local   ispos         ;This is necessary if you have any
;                                       ; labels in the %do section. See
;                                       ; chapter six for an explanation.

                  cmp    ax, 0
                  jge    ispos
                  neg    ax
ispos:
#endcode


#code      ("ABS" Sequence 2 w/positive value")
%discount
                  mov    ax, 1          ;Our positive value
%do
                  local  ispos

                  neg    ax
                  jns    ispos
                  neg    ax
ispos:
#endcode


#code      ("ABS" Sequence 3 w/positive value")
%discount
                  mov    ax, 1          ;Our positive value
%do
                  cwd
                  xor    ax, dx
                  and    dx, 1
                  add    ax, dx
#endcode


#code      ("ABS" Sequence 1 w/negative value")
%discount
                  mov    ax, -1         ;Our negative value
%do
                  local  ispos

                  cmp    ax, 0
                  jge    ispos
                  neg    ax
ispos:
#endcode


#code      ("ABS" Sequence 2 w/negative value")
%discount
                  mov    ax, -1         ;Our negative value
%do
                  local  ispos

                  neg    ax
                  jns    ispos
                  neg    ax
ispos:
#endcode


#code      ("ABS" Sequence 3 w/negative value")
%discount
                  mov    ax, -1         ;Our negative value
%do
                  cwd
                  xor    ax, dx
                  and    dx, 1
                  add    ax, dx
#endcode
#end
```

For your lab report: Describe which instruction sequence is fastest on your machine. Try to explain the results.

For additional credit: Try this code on different 80x86 CPUs (if available) and compare the results.

❏ Exercise 8: Run the sample IBM/L files found on the diskette accompanying this lab manual (LAB8a_5.IBM, etc.)

For your lab report: Explain the results. Describe which instruction sequences are most efficient in each group.

For additional credit: Devise your own code sequences and run them under IBM/L to determine which of several code sequences is the fastest.

❏ Exercise 9: Run the CPUDYNO program on your system.

For your lab report: compare the results to the published timings for various 8086 instructions. Discuss what could cause the differences.

For additional credit: run CPUDYNO on several different Intel CPUs. Compare their timings against each other and the published results (note: there are several files containing results for various Intel CPUs on the diskette).

❏ Exercise 10: For this exercise, assemble the “LAB10_5.ASM” file on the diskette accompanying this lab manual using the command “ml /Fi lab10_5.asm” to produce an .exe value you can load into CodeView. Connect the circuit you built in the laboratory exercises for Chapter Two to the printer port (LPT1:). Then, following the comments in the code below, single step through this program and observe the results.

```
; LAB10_5.asm
; Sample program that demonstrates the use of the IN & OUT instructions.

dseg              segment para public 'data'

; The port variable holds the base address of the LPT1: parallel printer
; port.  This is the address of the output port that controls the LEDs
; on the lab circuitry.

Port              word   ?

; InPort is the address of the LPT1: input port.  You can read the switches
; on the lab circuitry from this port.

InPort            word   ?

dseg              ends

cseg              segment para public 'code'
                  assume cs:cseg, ds:dseg

Main              proc
                  mov    ax, dseg
                  mov    ds, ax

; The base address of the LPT1: port is stored at address 40:8 in memory.
; Fetch that value and initialize our Port and InPort variables with that
; value.  (Note: if you want to use LPT2: or LPT3: rather than LPT1:, their
; base addresses appear at locations 40:A and 40:C, respectively.)

                  mov    ax, 40h       ;Point ES at the BIOS data segment.
                  mov    es, ax

; Change the following value from 8 to 0ah or 0ch for LPT2: or LPT3:
```

```
                mov     ax, es:[8];Fetch base address of LPT1: port.
                mov     Port, ax
                inc     ax              ;Point at input port.
                mov     InPort, ax

; The switches on your lab circuitry come in on bits 3, 4, 5, and 6 of
; the input port.  These bits contain a one if the switch is in the *OFF*
; position.  They contain a zero if the switch is in the *ON* position.
; Intuitively, this is backwards.  Be aware of this.
;
; Using CodeView, single step through the following instructions.
; After the execution of each IN instruction, change the switch settings
; on your circuitry and observe the value read into AL with the execution
; of each successive IN statement.

                mov     dx, InPort
                in      al, dx
                in      al, dx
                in      al, dx
                in      al, dx
                in      al, dx
                in      al, dx
                in      al, dx
                in      al, dx

; The LEDs on your lab circuitry are connected to bits zero through eight
; of the output port (Port).  Writing a one to a particular bit turns that
; LED *ON*.  Writing a zero to a particular bit turns that LED off.  Note
; that address "Port" is an I/O address.  You can read and write the data
; at that address (reading this port reads the last value written to it).
; The following code reads the switches, inverts their values, and then
; initializes LEDs zero through three with these switch values.  The code
; then rotates these bit settings through the LEDs several times.  Use
; CodeView to single step through these instructions.

                in      al, dx          ;DX still contains InPort's value.
                mov     cl, 3           ;Move the switches down to bit 0.
                shr     al, cl
                and     al, 0Fh         ;Mask out the other bits.
                xor     al, 0Fh         ;Invert switch readings.

                mov     dx, Port;Point DX out the output port.
                out     dx, al          ;Write switch settings to the LEDs.

                in      al, dx          ;Read previous value.
                rol     al, 1           ;Rotate bits.
                out     dx, al          ;Write back to LEDs.

                in      al, dx
                rol     al, 1
                out     dx, al

                in      al, dx
                rol     al, 1
                out     dx, al

                in      al, dx
                rol     al, 1
                out     dx, al

                in      al, dx
                rol     al, 1
                out     dx, al

                in      al, dx
                rol     al, 1
                out     dx, al
```

```
                    in      al, dx
                    rol     al, 1
                    out     dx, al

                    in      al, dx
                    rol     al, 1
                    out     dx, al

Quit:               mov     ah, 4ch                 ;DOS opcode to quit program.
                    int     21h                     ;Call DOS.
Main                endp
cseg                ends

sseg                segment para stack 'stack'
stk                 byte    1024 dup ("stack   ")
sseg                ends

zzzzzzseg           segment para public 'zzzzzz'
LastBytes           byte    16 dup (?)
zzzzzzseg           ends
                    end     Main
```

For your lab report: Describe what happens to AX, DX and the LEDs after the execution of each instruction above.

6.19 Sample Programs

Most real programs only use a small subset of the entire 80x86 instruction set. Therefore you do not have to learn the complete operation and usage of *every* 80x86 instruction in order to write typical assembly language programs. However, if you are comfortable with the entire instruction set, you will probably write better assembly language programs because you'll often use an appropriate instruction sequence rather than synthesize the operation with a longer sequence of instructions.

This section presents several short sample programs that demonstrate the use of several 80x86 assembly language instructions. It shows the syntax of many instructions and how one would typically use them.

6.19.1 Sample Program #1: Simple Arithmetic

This program demonstrates how to use the 80x86 add, sub, neg, inc, and dec instructions to perform simple calculations.

```
; Simple Arithmetic
; This program demonstrates some simple arithmetic instructions.

              .386
              option     segment:use16

dseg          segment    para public 'data'

; Some type definitions for the variables we will declare:

uint          typedef    word             ;Unsigned integers.
integer       typedef    sword            ;Signed integers.


; Some variables we can use:

j             integer    ?
k             integer    ?
l             integer    ?

u1            uint       ?
u2            uint       ?
u3            uint       ?

dseg          ends

cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

Main          proc
              mov        ax, dseg
              mov        ds, ax
              mov        es, ax

; Initialize our variables:

              mov        j, 3
              mov        k, -2

              mov        u1, 254
              mov        u2, 22

; Compute L := j+k and u3 := u1+u2
```

```
                mov         ax, J
                add         ax, K
                mov         L, ax

                mov         ax, u1              ;Note that we use the "ADD"
                add         ax, u2              ; instruction for both signed
                mov         u3, ax              ; and unsigned arithmetic.

; Compute L := j-k and u3 := u1-u2

                mov         ax, J
                sub         ax, K
                mov         L, ax

                mov         ax, u1              ;Note that we use the "SUB"
                sub         ax, u2              ; instruction for both signed
                mov         u3, ax              ; and unsigned arithmetic.

; Compute L := -L

                neg         L

; Compute L := -J

                mov         ax, J               ;Of course, you would only use the
                neg         ax                  ; NEG instruction on signed values.
                mov         L, ax

; Compute K := K + 1 using the INC instruction.

                inc         K

; Compute u2 := u2 + 1 using the INC instruction.
; Note that you can use INC for signed and unsigned values.

                inc         u2

; Compute J := J - 1 using the DEC instruction.

                dec         J

; Compute u2 := u2 - 1 using the DEC instruction.
; Note that you can use DEC for signed and unsigned values.

                dec         u2


Quit:           mov         ah, 4ch             ;DOS opcode to quit program.
                int         21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

6.19.2 Sample Program #2: Multiplication and Division

Unfortunately, multiplication and division work a little differently on the 80x86 than the other arithmetic instructions like add and subtract. This section demonstrates the use the mul, imul, div, and idiv instructions in an assembly language program. Don't forget that the division operation can crash your program if you attempt to divide by zero or if the operands are otherwise out of range.

```
; Simple Arithmetic: multiplication and division
; This program demonstrates some simple arithmetic instructions.

              .386                        ;So we can use extended registers
              option     segment:use16    ; and addressing modes.

dseg          segment    para public 'data'

; Some type definitions for the variables we will declare:

uint          typedef    word             ;Unsigned integers.
integer       typedef    sword            ;Signed integers.


; Some variables we can use:

j             integer    ?
k             integer    ?
l             integer    ?

u1            uint       ?
u2            uint       ?
u3            uint       ?

dseg          ends

cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

Main          proc
              mov        ax, dseg
              mov        ds, ax
              mov        es, ax

; Initialize our variables:

              mov        j, 3
              mov        k, -2

              mov        u1, 254
              mov        u2, 22


; Extended multiplication using 8086 instructions.
;
; Note that there are separate multiply instructions for signed and
; unsigned operands.
;
; L := J * K (ignoring overflow)

              mov        ax, J
              imul       K                ;Computes DX:AX := AX * K
              mov        L, ax            ;Ignore overflow into DX.
```

```
; u3 := u1 * u2

              mov       ax, u1
              mul       u2                 ;Computes DX:AX := AX * U2
              mov       u3, ax             ;Ignore overflow in DX.


; Extended division using 8086 instructions.
;
; Like multiplication, there are separate instructions for signed
; and unsigned operands.
;
; It is absolutely imperative that these instruction sequences sign
; extend or zero extend their operands to 32 bits before dividing.
; Failure to do so will may produce a divide error and crash the
; program.
;
; L := J div K

              mov       ax, J
              cwd                          ;*MUST* sign extend AX to DX:AX!
              idiv      K                  ;AX := DX:AX/K, DX := DX:AX mod K
              mov       L, ax

; u3 := u1/u2

              mov       ax, u1
              mov       dx, 0              ;Must zero extend AX to DX:AX!
              div       u2                 ;AX := DX:AX/u2, DX := DX:AX mod u2
              mov       u3, ax

; Special forms of the IMUL instruction available on 80286, 80386, and
; later processors.  Technically, these instructions operate on signed
; operands only, however, they do work fine for unsigned operands as well.
; Note that these instructions produce a 16-bit result and set the overflow
; flag if overflow occurs.
;
; L := J * 10 (80286 and later only)

              imul      ax, J, 10          ;AX := J*10
              mov       L, ax

; L := J * K (80386 and later only)

              mov       ax, J
              imul      ax, K
              mov       L, ax



Quit:         mov       ah, 4ch            ;DOS opcode to quit program.
              int       21h                ;Call DOS.
Main          endp

cseg          ends

sseg          segment   para stack 'stack'
stk           byte      1024 dup ("stack   ")
sseg          ends

zzzzzzseg     segment   para public 'zzzzzz'
LastBytes     byte      16 dup (?)
zzzzzzseg     ends
```

```
                end         Main
```

6.19.3 Sample Program #3: Logical Operations

The following short assembly language program demonstrates the use of the and, or, xor, and not instructions.

```
; Logical Operations
; This program demonstrates the AND, OR, XOR, and NOT instructions

                .386                        ;So we can use extended registers
                option      segment:use16    ; and addressing modes.

dseg            segment     para public 'data'


; Some variables we can use:

j               word        0FF00h
k               word        0FFF0h
l               word        ?

c1              byte        'A'
c2              byte        'a'

LowerMask       byte        20h

dseg            ends


cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg
                mov         ds, ax
                mov         es, ax

; Compute L := J and K (bitwise AND operation):

                mov         ax, J
                and         ax, K
                mov         L, ax

; Compute L := J or K (bitwise OR operation):

                mov         ax, J
                or          ax, K
                mov         L, ax

; Compute L := J xor K (bitwise XOR operation):

                mov         ax, J
                xor         ax, K
                mov         L, ax

; Compute L := not L (bitwise NOT operation):

                not         L
```

```
; Compute L := not J (bitwise NOT operation):

              mov       ax, J
              not       ax
              mov       L, ax

; Clear bits 0..3 in J:

              and       J, 0FFF0h

; Set bits 0..3 in K:

              or        K, 0Fh

; Invert bits 4..11 in L:

              xor       L, 0FF0h

; Convert the character in C1 to lower case:

              mov       al, c1
              or        al, LowerMask
              mov       c1, al

; Convert the character in C2 to upper case:

              mov       al, c2
              and       al, 5Fh           ;Clears bit 5.
              mov       c2, al



Quit:         mov       ah, 4ch           ;DOS opcode to quit program.
              int       21h               ;Call DOS.
Main          endp

cseg          ends

sseg          segment   para stack 'stack'
stk           byte      1024 dup ("stack   ")
sseg          ends

zzzzzzseg     segment   para public 'zzzzzz'
LastBytes     byte      16 dup (?)
zzzzzzseg     ends
              end       Main
```

6.19.4 Sample Program #4: Shift and Rotate Instructions

The following sample program shows how to use some of the 80x86's shift and rotate instructions. It also demonstrates how to pack data using these instructions.

```
; Shift and Rotate Instructions

              .386                        ;So we can use extended registers
              option    segment:use16     ; and addressing modes.

dseg          segment   para public 'data'
```

```
; The following structure holds the bit values for an 80x86 mod-reg-r/m byte.

mode            struct
modbits         byte        ?
reg             byte        ?
rm              byte        ?
mode            ends

Adrs1           mode        {11b, 100b, 111b}
modregrm        byte        ?

var1            word        1
var2            word        8000h
var3            word        0FFFFh
var4            word        ?

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg
                mov         ds, ax
                mov         es, ax

; Shifts and rotates directly on memory locations:
;
; var1 := var1 shl 1

                shl         var1, 1

; var1 := var1 shr 1

                shr         var1, 1

; On 80286 and later processors, you can shift by more than one bit at
; at time:

                shl         var1, 4
                shr         var1, 4

; The arithmetic shift right instruction retains the H.O. bit after each
; shift.  The following SAR instruction sets var2 to 0FFFFh

                sar         var2, 15

; On all processors, you can specify a shift count in the CL register.
; The following instruction restores var2 to 8000h:

                mov         cl, 15
                shl         var2, cl

; You can use the shift and rotate instructions, along with the logical
; instructions, to pack and unpack data.  For example, the following
; instruction sequence extracts bits 10..13 of var3 and leaves
; this value in var4:

                mov         ax, var3
                shr         ax, 10              ;Move bits 10..13 to 0..3.
                and         ax, 0Fh             ;Keep only bits 0..3.
                mov         var4, ax

; You can use the rotate instructions to compute this value somewhat faster
; on older processors like the 80286.

                mov       ax, var3
                rol       ax, 6              ;Six rotates rather than 10 shifts.
                and       ax, 0Fh
                mov       var4, ax

; You can use the shift and OR instructions to easily merge separate fields
; into a single value.  For example, the following code merges the mod, reg,
; and r/m fields (maintained in separate bytes) into a single mod-reg-r/m
; byte:


                mov       al, Adrs1.modbits
                shl       al, 3
                or        al, Adrs1.reg
                shl       al, 3
                or        al, Adrs1.rm
                mov       modregrm, al

; If you've only got and 8086 or 8088 chip, you'd have to use code like the
; following:

                mov       al, Adrs1.modbits ;Get mod field
                shl       al, 1
                shl       al, 1
                or        al, Adrs1.reg     ;Get reg field
                mov       cl, 3
                shl       al, cl            ;Make room for r/m field.
                or        al, Adrs1.rm      ;Merge in r/m field.
                mov       modregrm, al      ;Save result away.

Quit:           mov       ah, 4ch           ;DOS opcode to quit program.
                int       21h               ;Call DOS.
Main            endp

cseg            ends

sseg            segment   para stack 'stack'
stk             byte      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       Main
```

6.19.5 Sample Program #5: Bit Operations and the SETcc Instruction

The following program shows how to test to see if certain bits are set or clear. It also demonstrates the use of the set*cc* instructions to set a boolean variable true or false depending upon some condition.

```
; Bit Operations and SETcc Instructions

                .386                        ;So we can use extended registers
                option    segment:use16     ; and addressing modes.

dseg            segment   para public 'data'
```

```
; Some type definitions for the variables we will declare:

uint           typedef    word              ;Unsigned integers.
integer        typedef    sword             ;Signed integers.


; Some variables we can use:

j              integer    ?
k              integer    ?
u1             uint       2
u2             uint       2
Result         byte       ?

dseg           ends



cseg           segment    para public 'code'
               assume     cs:cseg, ds:dseg

Main           proc
               mov        ax, dseg
               mov        ds, ax
               mov        es, ax

; Initialize some variables

               mov        j, -2
               mov        k, 2

; The SETcc instructions store a one or zero into their operand if the
; specified condition is true or false, respectively.  The TEST instruction
; logically ANDs its operands and sets the flags accordingly (in particular,
; TEST sets/clears the zero flag if there is/isn't a zero result).  We can
; use these two facts to copy a single bit (zero extended) to a byte operand.

               test       j, 11000b         ;Test bits 4 and 5.
               setne      Result            ;Result=1 if bits 4 or 5 of J are 1.

               test       k, 10b            ;Test bit #1.
               sete       Result            ;Result=1 if bit #1 = 0.

; The SETcc instructions are particularly useful after a CMP instruction.
; You can set a boolean value according to the result of the comparison.
;
; Result := j <= k

               mov        ax, j
               cmp        ax, k
               setle      Result            ;Note that "setle" is for signed values.

; Result := u1 <= u2

               mov        ax, u1
               cmp        ax, u2
               setbe      Result            ;Note that "setbe" is for unsigned values.

; One thing nice about the boolean results that the SETcc instructions
; produce is that we can AND, OR, and XOR them and get the same results
; one would expect in a HLL like C, Pascal, or BASIC.
;
; Result := (j < k) and (u1 > u2)
```

```
                mov         ax, j
                cmp         ax, k
                setl        bl                  ;Use "setl" for signed comparisons.

                mov         ax, u1
                cmp         ax, u2
                seta        al                  ;Use "seta" for unsigned comparisons.

                and         al, bl              ;Logically AND the two boolean results
                mov         Result, al          ; and store the result away.

; Sometimes you can use the shift and rotate instructions to test to see
; if a specific bit is set.  For example, SHR copies bit #0 into the carry
; flag and SHL copies the H.O. bit into the carry flag.  We can easily test
; these bits as follows:
;
; Result := bit #15 of J.

                mov         ax, j
                shl         ax, 1
                setc        Result

; Result := bit #0 of u1:

                mov         ax, u1
                shr         ax, 1
                setc        Result

; If you don't have an 80386 or later processor and cannot use the SETcc
; instructions, you can often simulate them.  Consider the above two
; sequences rewritten for the 8086:

;
; Result := bit #15 of J.

                mov         ax, j
                rol         ax, 1               ;Copy bit #15 to bit #0.
                and         al, 1               ;Strip other bits.
                mov         Result, al

; Result := bit #0 of u1:

                mov         ax, u1
                and         al, 1               ;Strip unnecessary bits.
                mov         Result, al

Quit:           mov         ah, 4ch             ;DOS opcode to quit program.
                int         21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

6.19.6 Sample Program #6: String Instructions

This program demonstrates simple uses of some string instructions. It shows how you can easily manipulate array elements using these instructions.

```
; String Instructions

                .386                            ;So we can use extended registers
                option      segment:use16       ; and addressing modes.

dseg            segment     para public 'data'

String1         byte        "String",0
String2         byte        7 dup (?)

Array1          word        1, 2, 3, 4, 5, 6, 7, 8
Array2          word        8 dup (?)

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg
                mov         ds, ax
                mov         es, ax


; The string instructions let you easily copy data from one array to
; another.  If the direction flag is clear, the movsb instruction
; does the equivalent of the following:
;
;       mov es:[di], ds:[si]
;       inc    si
;       inc    di
;
; The following code copies the seven bytes from String1 to String2:

                cld                             ;Required if you want to INC SI/DI

                lea         si, String1
                lea         di, String2

                movsb                           ;String2[0] := String1[0]
                movsb                           ;String2[1] := String1[1]
                movsb                           ;String2[2] := String1[2]
                movsb                           ;String2[3] := String1[3]
                movsb                           ;String2[4] := String1[4]
                movsb                           ;String2[5] := String1[5]
                movsb                           ;String2[6] := String1[6]

; The following code sequence demonstrates how you can use the LODSW and
; STOWS instructions to manipulate array elements during the transfer.
; The following code computes
;
;       Array2[0] := Array1[0]
;       Array2[1] := Array1[0] * Array1[1]
;       Array2[2] := Array1[0] * Array1[1] * Array1[2]
;       etc.
;
```

```
; Of course, it would be far more efficient to put the following code
; into a loop, but that will come later.

              lea       si, Array1
              lea       di, Array2

              lodsw
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw

              lodsw
              imul      ax, dx
              mov       dx, ax
              stosw



Quit:         mov       ah, 4ch           ;DOS opcode to quit program.
              int       21h               ;Call DOS.
Main          endp

cseg          ends

sseg          segment   para stack ‘stack’
stk           byte      1024 dup (“stack   “)
sseg          ends

zzzzzzseg     segment   para public ‘zzzzzz’
LastBytes     byte      16 dup (?)
zzzzzzseg     ends
              end       Main
```

6.19.7 Sample Program #7: Unconditional JMP Instructions

This program demonstrates various forms of the unconditional jump instruction. To fully appreciate the differences, you will need to look at the object code produced by this program. You can do this by assembling the program and producing an assembly listing. The MASM command to do this is[6]

ml /Fl ex7_5.asm

This produces a file named “ex7_5.lst” that contains the assembled listing. The assembly listing contains the offsets and opcodes for each instruction. By studying these opcodes you can see the differences between these instructions. See Appendix D for details on **jmp** instruction encoding.

```
; Unconditional Jumps

               .386
               option     segment:use16

dseg           segment    para public ‘data’


; Pointers to statements in the code segment

IndPtr1        word       IndTarget2
IndPtr2        dword      IndTarget3



dseg           ends


cseg           segment    para public ‘code’
               assume     cs:cseg, ds:dseg

Main           proc
               mov        ax, dseg
               mov        ds, ax
               mov        es, ax

; JMP instructions transfer control to the
; location specified in the operand field.
; This is typically a label that appears
; in the program.
;
; There are many variants of the JMP
; instruction.  The first is a two-byte
; opcode that transfers control to +/-128
; bytes around the current instruction:

               jmp        CloseLoc
               nop
CloseLoc:


; The next form is a three-byte instruction
; that allows you to jump anywhere within
; the current code segment.  Normally, the
; assembler would pick the shortest version
; of a given JMP instruction, the “near ptr”
```

6. The file for this program, ex7_5.asm, is on the diskette accompanying this laboratory manual.

```
; operand on the following instruction
; forces a near (three byte) JMP:


              jmp       near ptr NearLoc
              nop
NearLoc:


; The third form to consider is a five-byte
; instruction that provides a full segmented
; address operand.  This form of the JMP
; instruction lets you transfer control any-
; where in the program, even to another
; segment.  The “far ptr” operand forces
; this form of the JMP instruction:

              jmp       far ptr FarLoc
              nop
FarLoc:


; You can also load the target address of a
; near JMP into a register and jump indirectly
; to the target location.  Note that you can
; use any 80x86 general purpose register to
; hold this address; you are not limited to
; the BX, SI, DI, or BP registers.

              lea       dx, IndTarget
              jmp       dx
              nop
IndTarget:


; You can even jump indirect through a memory
; variable.  That is, you can jump though a
; pointer variable directly without having to
; first load the pointer variable into a reg-
; ister (Chapter Six describes why the following
; labels need two colons).

              jmp       IndPtr1
              nop
IndTarget2::


; You can even execute a far jump indirect
; through memory.  Just specify a dword
; variable in the operand field of a JMP
; instruction:

              jmp       IndPtr2
              nop
IndTarget3::



Quit:         mov       ah, 4ch
              int       21h
Main          endp

cseg          ends
```

```
sseg            segment    para stack 'stack'
stk             byte       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

6.19.8 Sample Program #8: CALL and Interrupt Instructions

This program demonstrates various forms of the `call`, `ret`, and `int` instructions. Like the previous program, you will need to look at the object code and Appendix D to truly appreciate the differences in these instructions. To produce an assembly listing, use the following command on the ex8_5.asm file:

ml /Fl ex8_5.asm

This produces the file "ex8_5.lst" that contains a listing of the bytes each instruction emits.

Chapter Six discusses the use of the **proc** and **endp** statements. This program uses them to create near and far procedures that this code can invoke using the `call` instruction.

```
; CALL and INT Instructions

                .386
                option     segment:use16

dseg            segment    para public 'data'

; Some pointers to our subroutines:

SPtr1           word       Subroutine1
SPtr2           dword      Subroutine2

dseg            ends


cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Subroutine1     proc       near
                ret
Subroutine1     endp

Subroutine2     proc       far
                ret
Subroutine2     endp


Main            proc
                mov        ax, dseg
                mov        ds, ax
                mov        es, ax

; Near call:

                call       Subroutine1

; Far call:
```

```
                call       Subroutine2

; Near register-indirect call:

                lea        cx, Subroutine1
                call       cx

; Near memory-indirect call:

                call       SPtr1

; Far memory-indirect call:

                call       SPtr2

; INT transfers control to a routine whose
; address appears in the interrupt vector
; table (see Chapter 15 for details on
; the interrupt vector table). The following
; call tells the PC’s BIOS to print the
; ASCII character in AL to the display.

                mov        ah, 0eh
                mov        al, ‘A’
                int        10h

; INTO generates an INT 4 if the 80x86
; overflow flag is set.  It becomes a
; NOP if the overflow flag is clear.
; You can use this instruction after
; an arithmetic operation to quickly
; test for a fatal overflow.  Note:
; the following sequence does *not*
; generate an overflow.  Do not modify
; it so that it does unless you add an
; INT 4 interrupt service routine to
; the interrupt vector table (see Chapter
; 15 for details)

                mov        ax, 2
                add        ax, 4
                into

Quit:           mov        ah, 4ch
                int        21h
Main            endp
cseg            ends

sseg            segment    para stack ‘stack’
stk             byte       1024 dup (“stack   “)
sseg            ends

zzzzzzseg       segment    para public ‘zzzzzz’
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

6.19.9 Sample Program #9: Conditional Jump Instructions

The conditional jump instructions are some of the most important instructions in the 80x86 instruction set. An assembly language programmer uses these instructions to make decisions and create loops in a program. The following program demonstrates some of the capabilities of these instructions.

On the 80286 and earlier processors, the conditional instructions only allow you to branch to an instruction ±128 bytes around the current instruction. With MASM 5.1 and earlier, you had to manually convert out of range branches. MASM 6.x and later will do this for you automatically. The 80386 and later processor provide special forms of the conditional jump instructions that allow you to jump anywhere within the current code segment. If you specify the use of an 80386 or later processor, MASM will automatically use this form of the instruction if the shorter form is out of range. The following program demonstrates how MASM 6.x automatically extends out of range jumps. Once again, you will need to produce an assembly listing to see the actual opcodes MASM emits. To do this, use the DOS command:

```
ml /Fl ex9_5.asm
```

This produces the file “ex9_5.lst” that contains the assembly listing.

Do not run this program. It’s only purpose is to demonstrate how MASM emits opcodes for various conditional jump instructions. It may hang the machine if you attempt to run it from DOS.

```
; Conditional JMP Instructions, Part I

              .386
              option     segment:use16
dseg          segment    para public ‘data’
J             sword      ?
K             sword      ?
L             sword      ?
dseg          ends



cseg          segment    para public ‘code’
              assume     cs:cseg, ds:dseg

Main          proc
              mov        ax, dseg
              mov        ds, ax
              mov        es, ax


; 8086 conditional jumps are limited to
; +/- 128 bytes because they are only
; two bytes long (one byte opcode, one
; byte displacement).

              .8086
              ja         lbl
              nop
lbl:


; MASM 6.x will automatically extend out of
; range jumps.  The following are both
; equivalent:

              ja         lbl2
              byte       150 dup (0)       ;Make target greater than 128 bytes away.
lbl2:
```

```
              jna       Temp
              jmp       lbl3
Temp:
              byte      150 dup (0)       ;Make target greater than 128 bytes away.
lbl3:


; The 80386 and later processors support a
; special form of the conditional jump
; instructions that allow a two-byte displace-
; ment, so MASM 6.x will assemble the code
; to use this form if you've specified an
; 80386 processor.

              .386
              ja        lbl4
              byte      150 dup (0)       ;Make target more than 128 bytes away.
lbl4:


; The conditional jump instructions work
; well with the CMP instruction to let you
; execute certain instruction sequences
; only if a condition is true or false.
;
; if (J <= K) then
;      L := L + 1
; else  L := L - 1

              mov       ax, J
              cmp       ax, K
              jnle      DoElse
              inc       L
              jmp       ifDone

DoElse:       dec       L
ifDone:


; You can also use a conditional jump to
; create a loop in an assembly language
; program:
;
; while (j >= k) do begin
;
;      j := j - 1;
;      k := k + 1;
;      L := j * k;
; end;

WhlLoop:      mov       ax, j
              cmp       ax, k
              jnge      QuitLoop

              dec       j
              inc       k
              mov       ax, j
              imul      ax, k
              mov       L, ax
              jmp       WhlLoop

QuitLoop:

Quit:         mov       ah, 4ch           ;DOS opcode to quit program.
```

```
                int       21h              ;Call DOS.
Main            endp
cseg            ends

sseg            segment   para stack 'stack'
stk             byte      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end       Main
```

6.19.10Sample Program #10: More Conditional Jump Instructions

This short sample program demonstrates how to use the loop and loopne instructions to create simple loops within your programs. It also demonstrates how to use the conditional jump instructions to simulate the operation of the set*cc* instructions on 80286 and earlier processors.

```
; Conditional JMP Instructions, Part II

                .386
                option    segment:use16
dseg            segment   para public 'data'

Array1          word      1, 2, 3, 4, 5, 6, 7, 8
Array2          word      8 dup (?)

String1         byte      "This string contains lower case characters",0
String2         byte      128 dup (0)

j               sword     5
k               sword     6

Result          byte      ?

dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

Main            proc
                mov       ax, dseg
                mov       ds, ax
                mov       es, ax

; You can use the LOOP instruction to repeat a sequence of statements
; some specified number of times in an assembly language program.
; Consider the code taken from EX6_5.ASM that used the string
; instructions to produce a running product:
;
; The following code uses a loop instruction to compute:
;
;     Array2[0] := Array1[0]
;     Array2[1] := Array1[0] * Array1[1]
;     Array2[2] := Array1[0] * Array1[1] * Array1[2]
;     etc.

                cld
```

```
              lea       si, Array1
              lea       di, Array2
              mov       dx, 1                ;Initialize for 1st time.
              mov       cx, 8                ;Eight elements in the arrays.

LoopHere:     lodsw
              imul      ax, dx
              mov       dx, ax
              stosw
              loop      LoopHere


; The LOOPNE instruction is quite useful for controlling loops that
; stop on some condition or when the loop exceeds some number of
; iterations.  For example, suppose string1 contains a sequence of
; characters that end with a byte containing zero.  If you wanted to
; convert those characters to upper case and copy them to string2,
; you could use the following code.  Note how this code ensures that
; it does not copy more than 127 characters from string1 to string2
; since string2 only has enough storage for 127 characters (plus a
; zero terminating byte).

              lea       si, String1
              lea       di, String2
              mov       cx, 127              ;Max 127 chars to string2.

CopyStrLoop:  lodsb                          ;Get char from string1.
              cmp       al, 'a'              ;See if lower case
              jb        NotLower             ;Characters are unsigned.
              cmp       al, 'z'
              ja        NotLower
              and       al, 5Fh              ;Convert lower->upper case.
NotLower:
              stosb
              cmp       al, 0                ;See if zero terminator.
              loopne    CopyStrLoop          ;Quit if al or cx = 0.



; If you do not have an 80386 (or later) CPU and you would like the
; functionality of the SETcc instructions, you can easily achieve
; the same results using code like the following:
;
; Result := J <= K;

              mov       Result, 0            ;Assume false.
              mov       ax, J
              cmp       ax, K
              jnle      Skip1
              mov       Result, 1            ;Set to 1 if J <= K.
Skip1:


; Result := J = K;

              mov       Result, 0            ;Assume false.
              mov       ax, J
              cmp       ax, K
              jne       Skip2
              mov       Result, 1
Skip2:
```

```
Quit:          mov        ah, 4ch          ;DOS opcode to quit program.
               int        21h              ;Call DOS.
Main           endp

cseg           ends

sseg           segment    para stack ‘stack’
stk            byte       1024 dup (“stack   “)
sseg           ends

zzzzzzseg      segment    para public ‘zzzzzz’
LastBytes      byte       16 dup (?)
zzzzzzseg      ends
               end        Main
```

6.20 Programming Projects

- ❏ Program #1: Write a short "GetLine" routine which reads up to 80 characters from the user and places these characters in successive locations in a buffer in your data segment. Use the INT 16h and INT 10h system BIOS calls described in this chapter to input and output the characters typed by the user. Terminate input upon encountering a carriage return (ASCII code 0Dh) or after the user types the 80th character. Be sure to count the number of characters actually typed by the user for later use. There is a "shell" program specifically designed for this project on the diskette accompanying this manual (proj1_5.asm).

- ❏ Program #2: Modify the above routine so that it properly handles the backspace character (ASCII code 08h). Whenever the user presses a backspace key, you should remove the previous keystroke from the input buffer (unless there are no previous characters in the input buffer, in which case you ignore the backspace).

- ❏ Program #3: You can use the XOR operation to *encrypt* and *decrypt* data. If you XOR all the characters in a message with some value you will effectively *scramble* that message. You can retrieve the original message by XOR'ing the characters in the message with the same value again. Modify the code in Program #2 so that it encrypts each byte in the message with the value 0Fh and displays the encrypted message to the screen. After displaying the message, decrypt it by XOR'ing with 0Fh again and display the decrypted message. Note that you should use the count value computed by the "GetLine" code to determine how many characters to process.

- ❏ Program #4: Write a "PutString" routine that prints the characters pointed at by the es:di register pair. This routine should print all characters up to (but not including) a zero terminating byte. This routine should preserve all registers it modifies. There is a "shell" program specifically designed for this project on the diskette accompanying this manual (proj4_5.asm).

- ❏ Program #5: To output a 16-bit integer value as the corresponding string of decimal digits, you can use the following algorithm:

```
if value = 0 then write('0')
else begin

        DivVal := 10000;
        while (Value mod DivVal) = 0 do begin

                Value := Value mod DivVal;
                DivVal := DivVal div 10;

        end;

        while (DivVal > 1) do begin

                write ( chr( Value div DivVal + 48)); (* 48 = '0' *)
                Value := Value mod DivVal;
                DivVal := DivVal div 10;

        end;
end;
```

  Provide a short routine that takes an arbitrary value in ax and outputs it as the corresponding decimal string. Use the int 10h instruction to output the characters to the display. You can use the "shell" program provided on the diskette to begin this project (proj5_5.asm).

- ❏ Program #6: To input a 16-bit integer from the keyboard, you need to use code that uses the following algorithm:

```
        Value := 0
        repeat

                getchar(ch);
                if (ch >= '0') and (ch <= '9') then begin
```

```
                        Value := Value * 10 + ord(ch) - ord('0');
                    end;
                until (ch < '0') or (ch > '9');
```

Use the INT 16h instruction (described in this chapter) to read characters from the keyboard. Use the output routine in program #4 to display the input result. You can use the "shell" file proj6_5.asm to start this project.

- ❏ Program #7: Write a program that reads the switches from your circuitry from this lab. It should read the switches on the input port and write the switch values to four separate byte variables: sw1, sw2, sw3, and sw4. The variables should contain a one if the switch is in the on position, they should contain a zero if the switch is in the off position. Next, write a short routine that merges bit #0 in the four byte variables out1, out2, out3, and out4 together and writes them to LEDs zero through four. In your main program, write some code that reads the switches, computes the following boolean functions, stores the function results in out1..out4, and then outputs these values to the LEDs.

  Boolean Functions to support:

```
out1 = sw1•sw2 + sw3•sw4
out2 = sw1•sw2'•sw3•sw4' + sw1•sw2•sw3
out3 = sw1'•sw2'•sw3'•sw4' + sw1•sw2•sw3•sw4 + sw1'•sw2•sw3•sw4' + sw1•sw2'•sw3'•sw4
out4 = sw1 + sw2 + sw3'•sw4'
```

  You can use the proj7_5.asm file as a "shell" for this project.

- ❏ Program #8: The file "proj8_5.asm" on the diskette accompanying this lab manual is a short program that creates a "light show" on the circuit you constructed in the Chapter Two laboratory. Modify this file to produce a light show of your own. Some suggestions: by calling the delay routine several times in a row you can lengthen the time that one pattern appears on your LEDs. If you adjust the logic of the program a little, you can use different delays between different patterns to improve your light show.

6.21 Solutions to Selected Exercises

1) One possible example (there are many) FF+FF =1FE. FF is the largest possible eight bit value. The sum of these two large values requires only nine bits to hold the result.

4) Possible ways:

```
add     bx, 2

inc     bx
inc     bx

lea     bx, 2[bx]

sub     bx, -2

clc
adc     bx, 2

stc
adc     bx, 1

lea     bx, 1[bx]
inc     bx

inc     bx
add     bx, 1

etc.
```

5a)

```
        jns     TempL1
        jmp     Label1
TempL1:
```

15) The sign never changes with SAR, so arithmetic overflow does not occur.

17) IMUL is necessary for operations like array subscript computations. IDIV isn't used as often as IMUL.

21a) Zero.

21d) Carry and zero.

22c) Sign and overflow.

22d) Sign, overflow, and zero.

24) One way to do it (most ways involve using several registers):

```
mov     ax, I
mov     bx, J
mov     I, bx
mov     J, ax
```

25) One way to do it:

```
mov     ax, I
xchg    ax, J
mov     I, ax
```

30) Copying the floating point status register bits into the 80x86 flags register.

34) Point BX at a table whose entries are equal to their index into the table except for indices 'a'..'z'. In these entries put the upper case characters.

46)

```
pop     mem32
jmp     mem32
```