The 80x86 Instruction Set Chapter Six

Until now, there has been little discussion of the instructions available on the 80x86 microprocessor. This chapter rectifies this situation. Note that this chapter is mainly for *reference*. It explains what each instruction does, it does not explain how to combine these instructions to form complete assembly language programs. The rest of this book will explain how to do that.

6.0 Chapter Overview

This chapter discusses the 80x86 real mode instruction set. Like any programming language, there are going to be several instructions you use all the time, some you use occasionally, and some you will rarely, if ever, use. This chapter organizes its presentation by instruction class rather than importance. Since beginning assembly language programmers do not have to learn the entire instruction set in order to write meaningful assembly language programs, you will probably not have to learn how every instruction operates. The following list describes the instructions this chapter discusses. A "•" symbol marks the important instructions in each group. If you learn only these instructions, you will probably be able to write any assembly language program you want. There are many additional instructions, especially on the 80386 and later processors. These additional instructions make assembly language programming easier, but you do not need to know them to begin writing programs.

80x86 instructions can be (roughly) divided into eight different classes:

1) Data movement instructions
   - mov, lea, les , push, pop, pushf, popf
2) Conversions
   - cbw, cwd, xlat
3) Arithmetic instructions
   - add, inc sub, dec, cmp, neg, mul, imul, div, idiv
4) Logical, shift, rotate, and bit instructions
   - and, or, xor, not, shl, shr, rcl, rcr
5) I/O instructions
   - in, out
6) String instructions
   - movs, stos, lods
7) Program flow control instructions
   - jmp, call, ret, conditional jumps
8) Miscellaneous instructions.
   - clc, stc, cmc

The following sections describe all the instructions in these groups and how they operate.

At one time a text such as this one would recommend against using the extended 80386 instruction set. After all, programs that use such instructions will not run properly on 80286 and earlier processors. Using these additional instructions could limit the number of machines your code would run on. However, the 80386 processor is on the verge of disappearing as this text is being written. You can safely assume that most systems will contain an 80386sx or later processor. This text often uses the 80386 instruction set in various example programs. Keep in mind, though, that this is only for convenience. There is no program that appears in this text that could not be recoded using only 8088 assembly language instructions.

A word of advice, particularly to those who learn only the instructions noted above: as you read about the 80x86 instruction set you will discover that the individual 80x86 instructions are not very complex and have simple semantics. However, as you approach

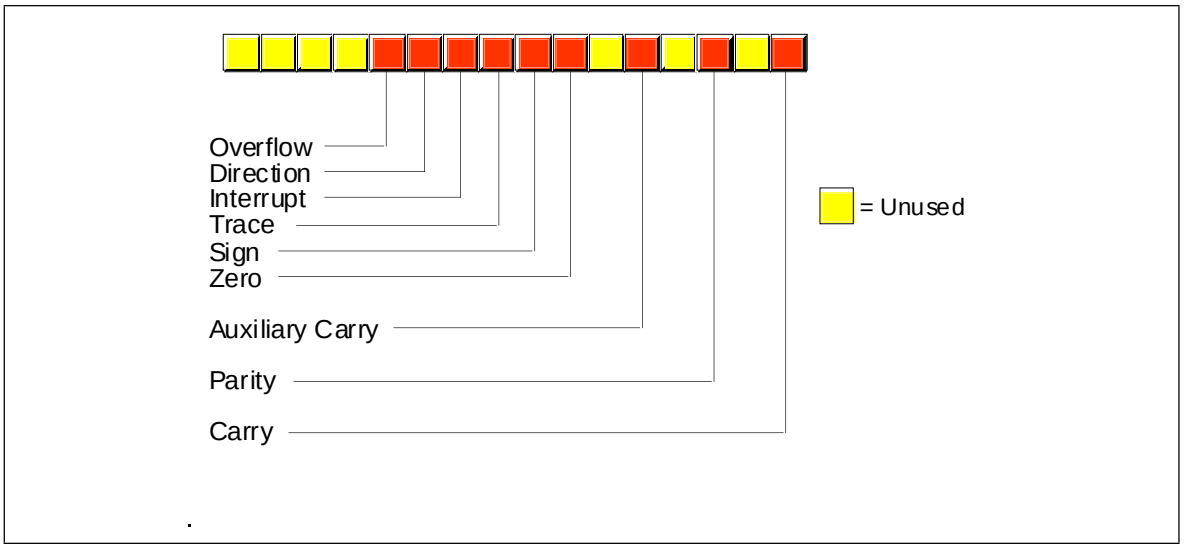

Figure 6.1 80x86 Flags Register

the end of this chapter, you may discover that you haven't got a clue how to put these simple instructions together to form a complex program. Fear not, this is a common problem. Later chapters will describe how to form complex programs from these simple instructions.

One quick note: this chapter lists many instructions as "available only on the 80286 and later processors." In fact, many of these instructions were available on the 80186 microprocessor as well. Since few PC systems employ the 80186 microprocessor, this text ignores that CPU. However, to keep the record straight...

6.1 The Processor Status Register (Flags)

The flags register maintains the current operating mode of the CPU and some instruction state information. Figure 6.1 shows the layout of the flags register.

The carry, parity, zero, sign, and overflow flags are special because you can test their status (zero or one) with the set*cc* and conditional jump instructions (see "The "Set on Condition" Instructions" on page 281 and "The Conditional Jump Instructions" on page 296). The 80x86 uses these bits, the *condition codes*, to make decisions during program execution.

Various arithmetic, logical, and miscellaneous instructions affect the *overflow flag*. After an arithmetic operation, this flag contains a one if the result does not fit in the signed destination operand. For example, if you attempt to add the 16 bit signed numbers 7FFFh and 0001h the result is too large so the CPU sets the overflow flag. If the result of the arithmetic operation does not produce a signed overflow, then the CPU clears this flag.

Since the logical operations generally apply to unsigned values, the 80x86 logical instructions simply clear the overflow flag. Other 80x86 instructions leave the overflow flag containing an arbitrary value.

The 80x86 string instructions use the *direction flag*. When the direction flag is clear, the 80x86 processes string elements from low addresses to high addresses; when set, the CPU processes strings in the opposite direction. See "String Instructions" on page 284 for additional details.

The *interrupt enable/disable flag* controls the 80x86's ability to respond to external events known as interrupt requests. Some programs contain certain instruction sequences that the CPU must not interrupt. The interrupt enable / disable flag turns interrupts on or off to guarantee that the CPU does not interrupt those critical sections of code.

The *trace flag* enables or disables the 80x86 trace mode. Debuggers (such as CodeView) use this bit to enable or disable the single step / trace operation. When set, the CPU interrupts each instruction and passes control to the debugger software, allowing the debugger to *single step* through the application. If the trace bit is clear, then the 80x86 executes instructions without the interruption. The 80x86 CPUs do not provide any instructions that directly manipulate this flag. To set or clear the trace flag, you must:

- Push the flags onto the 80x86 stack,
- Pop the value into another register,
- Tweak the trace flag value,
- Push the result onto the stack, and then
- Pop the flags off the stack.

If the result of some computation is negative, the 80x86 sets the *sign flag*. You can test this flag after an arithmetic operation to check for a negative result. Remember, a value is negative if its H.O. bit is one. Therefore, operations on unsigned values will set the sign flag if the result has a one in the H.O. position.

Various instructions set the *zero flag* when they generate a zero result. You'll often use this flag to see if two values are equal (e.g., after subtracting two numbers, they are equal if the result is zero). This flag is also useful after various logical operations to see if a specific bit in a register or memory location contains zero or one.

The *auxiliary carry flag* supports special binary coded decimal (BCD) operations. Since most programs don't deal with BCD numbers, you'll rarely use this flag and even then you'll not access it directly. The 80x86 CPUs do not provide any instructions that let you directly test, set, or clear this flag. Only the add, adc, sub, sbb, mul, imul, div, idiv, and BCD instructions manipulate this flag.

The *parity flag* is set according to the parity of the L.O. eight bits of any data operation. If an operation produces an even number of one bits, the CPU sets this flag. It clears this flag if the operation yields an odd number of one bits. This flag is useful in certain data communications programs, however, Intel provided it mainly to provide some compatibility with the older 8080 µP.

The *carry flag* has several purposes. First, it denotes an unsigned overflow (much like the overflow flag detects a signed overflow). You will also use it during multiprecision arithmetic and logical operations. Certain bit test, set, clear, and invert instructions on the 80386 directly affect this flag. Finally, since you can easily clear, set, invert, and test it, it is useful for various boolean operations. The carry flag has many purposes and knowing when to use it, and for what purpose, can confuse beginning assembly language programmers. Fortunately, for any given instruction, the meaning of the carry flag is clear.

The use of these flags will become readily apparent in the coming sections and chapters. This section is mainly a formal introduction to the individual flags in the register rather than an attempt to explain the exact function of each flag. For more details on the operation of each flag, keep reading...

6.2 Instruction Encodings

The 80x86 uses a binary encoding for each machine operation. While it is important to have a general understanding of how the 80x86 encodes instructions, it is not important that you memorize the encodings for all the instructions in the instruction set. If you were to write an assembler or disassembler (debugger), you would definitely need to know the exact encodings. For general assembly language programming, however, you won't need to know the exact encodings.

However, as you become more experienced with assembly language you will probably want to study the encodings of the 80x86 instruction set. Certainly you should be aware of such terms as *opcode, mod-reg-r/m byte, displacement value*, and so on. Although you do not need to memorize the parameters for each instruction, it is always a good idea to know the lengths and cycle times for instructions you use regularly since this will help

you write better programs. Chapter Three and Chapter Four provided a detailed look at instruction encodings for various instructions (80x86 and x86); such a discussion was important because you do need to understand how the CPU encodes and executes instructions. This chapter does not deal with such details. This chapter presents a higher level view of each instruction and assumes that you don't care how the machine treats bits in memory. For those few times that you will need to know the binary encoding for a particular instruction, a complete listing of the instruction encodings appears in Appendix D.

6.3 Data Movement Instructions

The data movement instructions copy values from one location to another. These instructions include mov, xchg, lds, lea, les, lfs, lgs, lss, push, pusha, pushad, pushf, pushfd, pop, popa, popad, popf, popfd, lahf, and sahf.

6.3.1 The MOV Instruction

The mov instruction takes several different forms:

```
mov     reg, reg[1]
mov     mem, reg
mov     reg, mem
mov     mem, immediate data
mov     reg, immediate data
mov     ax/al, mem
mov     mem, ax/al
mov     segreg, mem16
mov     segreg, reg16
mov     mem16, segreg
mov     reg16, segreg
```

The last chapter discussed the mov instruction in detail, only a few minor comments are worthwhile here. First, there are variations of the mov instruction that are faster and shorter than other mov instructions that do the same job. For example, both the mov ax, mem and mov reg, mem instructions can load the ax register from a memory location. On all processors the first version is shorter. On the earlier members of the 80x86 family, it is faster as well.

There are two very important details to note about the mov instruction. First, there is no memory to memory move operation. The mod-reg-r/m addressing mode byte (see Chapter Four) allows two register operands or a single register and a single memory operand. There is no form of the mov instruction that allows you to encode *two* memory addresses into the same instruction. Second, you cannot move immediate data into a segment register. The only instructions that move data into or out of a segment register have mod-reg-r/m bytes associated with them; there is no format that moves an immediate value into a segment register. Two common errors beginning programmers make are attempting a memory to memory move and trying to load a segment register with a constant.

The operands to the mov instruction may be bytes, words, or double words[2]. Both operands must be the same size or MASM will generate an error while assembling your program. This applies to memory operands and register operands. If you declare a variable, B, using byte and attempt to load this variable into the ax register, MASM will complain about a type conflict.

The CPU extends immediate data to the size of the destination operand (unless it is too big to fit in the destination operand, which is an error). Note that you *can* move an

1. This chapter uses "reg", by itself, to denote any eight bit, sixteen bit, or (on the 80386 and later) 32 bit general purpose CPU register (AL/AX/EAX, BL/BX/EBX, SI/ESI, etc.)
2. Double word operands are valid only on 80386 and later processors.

immediate value into a memory location. The same rules concerning size apply. However, MASM cannot determine the size of certain memory operands. For example, does the instruction `mov [bx], 0` store an eight bit, sixteen bit, or thirty-two bit value? MASM cannot tell, so it reports an error. This problem does *not* exist when you move an immediate value into a variable you've declared in your program. For example, if you've declared `B` as a byte variable, MASM knows to store an eight bit zero into `B` for the instruction `mov B, 0`. Only those memory operands involving pointers with no variable operands suffer from this problem. The solution is to explicitly tell MASM whether the operand is a byte, word, or double word. You can accomplish this with the following instruction forms:

```
mov     byte ptr [bx], 0
mov     word ptr [bx], 0
mov     dword ptr [bx], 0          (3)
```

(3) Available only on 80386 and later processors

For more details on the *type* ptr operator, see Chapter Eight.

Moves to and from segment registers are always 16 bits; the mod-reg-r/m operand must be 16 bits or MASM will generate an error. Since you cannot load a constant directly into a segment register, a common solution is to load the constant into an 80x86 general purpose register and then copy it to the segment register. For example, the following two instruction sequence loads the `es` register with the value 40h:

```
mov     ax, 40h
mov     es, ax
```

Note that almost any general purpose register would suffice. Here, `ax` was chosen arbitrarily.

The `mov` instructions do not affect any flags. In particular, the 80x86 preserves the flag values across the execution of a `mov` instruction.

6.3.2 The XCHG Instruction

The `xchg` (exchange) instruction swaps two values. The general form is

```
xchg    operand1, operand2
```

There are four specific forms of this instruction on the 80x86:

```
xchg    reg, mem
xchg    reg, reg
xchg    ax, reg16
xchg    eax, reg32    (3)
```

```
(3) Available only on 80386 and later processors
```

The first two general forms require two or more bytes for the opcode and mod-reg-r/m bytes (a displacement, if necessary, requires additional bytes). The third and fourth forms are special forms of the second that exchange data in the `(e)ax` register with another 16 or 32 bit register. The 16 bit form uses a single byte opcode that is shorter than the other two forms that use a one byte opcode and a mod-reg-r/m byte.

Already you should note a pattern developing: the 80x86 family often provides shorter and faster versions of instructions that use the `ax` register. Therefore, you should try to arrange your computations so that they use the `(e)ax` register as much as possible. The `xchg` instruction is a perfect example, the form that exchanges 16 bit registers is only one byte long.

Note that the order of the `xchg`'s operands does not matter. That is, you could enter `xchg mem, reg` and get the same result as `xchg reg, mem`. Most modern assemblers will automatically emit the opcode for the shorter `xchg ax, reg` instruction if you specify `xchg reg, ax`.

Both operands must be the same size. On pre-80386 processors the operands may be eight or sixteen bits. On 80386 and later processors the operands may be 32 bits long as well.

The xchg instruction does not modify any flags.

6.3.3 The LDS, LES, LFS, LGS, and LSS Instructions

The lds, les, lfs, lgs, and lss instructions let you load a 16 bit general purpose register and segment register pair with a single instruction. On the 80286 and earlier, the lds and les instructions are the only instructions that directly process values larger than 32 bits. The general form is

```
LxS     dest, source
```

These instructions take the specific forms:

```
lds     reg16, mem32
les     reg16, mem32
lfs     reg16, mem32            (3)
lgs     reg16, mem32            (3)
lss     reg16, mem32            (3)
```

(3) Available only on 80386 and later processors

Reg_{16} is any general purpose 16 bit register and mem_{32} is a double word memory location (declared with the dword statement).

These instructions will load the 32 bit double word at the address specified by mem_{32} into reg_{16} and the ds, es, fs, gs, or ss registers. They load the general purpose register from the L.O. word of the memory operand and the segment register from the H.O. word. The following algorithms describe the exact operation:

```
lds reg16, mem32:
                reg16 := [mem32]
                ds := [mem32 + 2]
les reg16, mem32:
                reg16 := [mem32]
                es := [mem32 + 2]
lfs reg16, mem32:
                reg16 := [mem32]
                fs := [mem32 + 2]
lgs reg16, mem32:
                reg16 := [mem32]
                gs := [mem32 + 2]
lss reg16, mem32:
                reg16 := [mem32]
                ss := [mem32 + 2]
```

Since the LxS instructions load the 80x86's segment registers, you must not use these instructions for arbitrary purposes. Use them to set up (far) pointers to certain data objects as discussed in Chapter Four. Any other use may cause problems with your code if you attempt to port it to Windows, OS/2 or UNIX.

Keep in mind that these instructions load the four bytes at a given memory location into the register pair; they do *not* load the address of a variable into the register pair (i.e., this instruction does not have an immediate mode). To learn how to load the address of a variable into a register pair, see Chapter Eight.

The LxS instructions do not affect any of the 80x86's flag bits.

6.3.4 The LEA Instruction

The lea (Load Effective Address) instruction is another instruction used to prepare pointer values. The lea instruction takes the form:

```
lea     dest, source
```

The specific forms on the 80x86 are

```
lea     reg16, mem
lea     reg32, mem              (3)
```

(3) Available only on 80386 and later processors.

It loads the specified 16 or 32 bit general purpose register with the *effective address* of the specified memory location. The effective address is the final memory address obtained after all addressing mode computations. For example, lea ax, ds:[1234h] loads the ax register with the address of memory location 1234h; here it just loads the ax register with the value 1234h. If you think about it for a moment, this isn't a very exciting operation. After all, the mov ax, immediate_data instruction can do this. So why bother with the lea instruction at all? Well, there are many other forms of a memory operand besides displacement-only operands. Consider the following lea instructions:

```
lea     ax, [bx]
lea     bx, 3[bx]
lea     ax, 3[bx]
lea     bx, 4[bp+si]
lea     ax, -123[di]
```

The lea ax, [bx] instruction copies the address of the expression [bx] into the ax register. Since the effective address is the value in the bx register, this instruction copies bx's value into the ax register. Again, this instruction isn't very interesting because mov can do the same thing, even faster.

The lea bx,3[bx] instruction copies the effective address of 3[bx] into the bx register. Since this effective address is equal to the current value of bx plus three, this lea instruction effectively adds three to the bx register. There is an add instruction that will let you add three to the bx register, so again, the lea instruction is superfluous for this purpose.

The third lea instruction above shows where lea really begins to shine. lea ax, 3[bx] copies the address of the memory location 3[bx] into the ax register; i.e., it adds three with the value in the bx register and moves the sum into ax. This is an excellent example of how you can use the lea instruction to do a mov operation and an addition with a single instruction.

The final two instructions above, lea bx,4[bp+si] and lea ax,-123[di] provide additional examples of lea instructions that are more efficient than their mov/add counterparts.

On the 80386 and later processors, you can use the scaled indexed addressing modes to multiply by two, four, or eight as well as add registers and displacements together. Intel *strongly* suggests the use of the lea instruction since it is much faster than a sequence of instructions computing the same result.

The (real) purpose of lea is to load a register with a memory address. For example, lea bx, 128[bp+di] sets up bx with the address of the byte referenced by 128[BP+DI]. As it turns out, an instruction of the form mov al,[bx] runs faster than an instruction of the form mov al,128[bp+di]. If this instruction executes several times, it is probably more efficient to load the effective address of 128[bp+di] into the bx register and use the [bx] addressing mode. This is a common optimization in high performance programs.

The lea instruction does not affect any of the 80x86's flag bits.

6.3.5 The PUSH and POP Instructions

The 80x86 push and pop instructions manipulate data on the 80x86's hardware stack. There are 19 varieties of the push and pop instructions[3], they are

3. Plus some synonyms on top of these 19.

```
push    reg16
pop     reg16
push    reg32                   (3)
pop     reg32                   (3)
push    segreg
pop     segreg                  (except CS)
push    memory
pop     memory
push    immediate_data          (2)
pusha                           (2)
popa                            (2)
pushad                          (3)
popad                           (3)
pushf
popf
pushfd                          (3)
popfd                           (3)
enter   imm, imm                (2)
leave                           (2)

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

The first two instructions push and pop a 16 bit general purpose register. This is a compact (one byte) version designed specifically for registers. Note that there is a second form that provides a mod-reg-r/m byte that could push registers as well; most assemblers only use that form for pushing the value of a memory location.

The second pair of instructions push or pop an 80386 32 bit general purpose register. This is really nothing more than the push register instruction described in the previous paragraph with a size prefix byte.

The third pair of push/pop instructions let you push or pop an 80x86 segment register. Note that the instructions that push fs and gs are longer than those that push cs, ds, es, and ss, see Appendix D for the exact details. You can only push the cs register (popping the cs register would create some interesting program flow control problems).

The fourth pair of push/pop instructions allow you to push or pop the contents of a memory location. On the 80286 and earlier, this must be a 16 bit value. For memory operations without an explicit type (e.g., [bx]) you must either use the pushw mnemonic or explicitly state the size using an instruction like push word ptr [bx]. On the 80386 and later you can push and pop 16 or 32 bit values[4]. You can use dword memory operands, you can use the pushd mnemonic, or you can use the dword ptr operator to force 32 bit operation. Examples:

```
push    DblWordVar
push    dword ptr [bx]
pushd   dword
```

The pusha and popa instructions (available on the 80286 and later) push and pop *all* the 80x86 16 bit general purpose registers. Pusha pushes the registers in the following order: ax, cx, dx, bx, sp, bp, si, and then di. Popa pops these registers in the reverse order. Pushad and Popad (available on the 80386 and later) do the same thing on the 80386's 32 bit register set. Note that these "push all" and "pop all" instructions do *not* push or pop the flags or segment registers.

The pushf and popf instructions allow you to push/pop the processor status register (the flags). Note that these two instructions provide a mechanism to modify the 80x86's trace flag. See the description of this process earlier in this chapter. Of course, you can set and clear the other flags in this fashion as well. However, most of the other flags you'll want to modify (specifically, the condition codes) provide specific instructions or other simple sequences for this purpose.

Enter and leave push/pop the bp register and allocate storage for local variables on the stack. You will see more on these instructions in a later chapter. This chapter does not con-

4. You can use the PUSHW and PUSHD mnemonics to denote 16 or 32 bit constant sizes.

sider them since they are not particularly useful outside the context of procedure entry and exit.

"So what do these instructions do?" you're probably asking by now. The push instructions move data onto the 80x86 hardware stack and the pop instructions move data from the stack to memory or to a register. The following is an algorithmic description of each instruction:

push instructions (16 bits):

```
        SP := SP - 2
        [SS:SP] := 16 bit operand (store result at location SS:SP.)
```

```
pop instructions (16 bits):
        16-bit operand := [SS:SP]
        SP := SP + 2
```

```
push instructions (32 bits):
        SP := SP - 4
        [SS:SP] := 32 bit operand
```

```
pop instructions (32 bits):
        32 bit operand := [SS:SP]
        SP := SP + 4
```

You can treat the pusha/pushad and popa/popad instructions as equivalent to the corresponding sequence of 16 or 32 bit push/pop operations (e.g., push ax, push cx, push dx, push bx, etc.).

Notice three things about the 80x86 hardware stack. First, it is always in the stack segment (wherever ss points). Second, the stack grows down in memory. That is, as you push values onto the stack the CPU stores them into successively lower memory locations. Finally, the 80x86 hardware stack pointer (ss:sp) always contains the address of the value on the top of the stack (the last value pushed on the stack).

You can use the 80x86 hardware stack for temporarily saving registers and variables, passing parameters to a procedure, allocating storage for local variables, and other uses. The push and pop instructions are extremely valuable for manipulating these items on the stack. You'll get a chance to see how to use them later in this text.

Most of the push and pop instructions do not affect any of the flags in the 80x86 processor status register. The popf/popfd instructions, by their very nature, can modify all the flag bits in the 80x86 processor status register (flags register). Pushf and pushfd push the flags onto the stack, but they do not change any flags while doing so.

All pushes and pops are 16 or 32 bit operations. There is no (easy) way to push a single eight bit value onto the stack. To push an eight bit value you would need to load it into the H.O. byte of a 16 bit register, push that register, and then add one to the stack pointer. On all processors except the 8088, this would slow future stack access since sp now contains an odd address, misaligning any further pushes and pops. Therefore, most programs push or pop 16 bits, even when dealing with eight bit values.

Although it is relatively safe to push an eight bit memory variable, be careful when popping the stack to an eight bit memory location. Pushing an eight bit variable with push word ptr ByteVar pushes two bytes, the byte in the variable ByteVar and the byte immediately following it. Your code can simply ignore the extra byte this instruction pushes onto the stack. Popping such values is not quite so straight forward. Generally, it doesn't hurt if you push these two bytes. However, it can be a disaster if you pop a value and wipe out the following byte in memory. There are only two solutions to this problem. First, you could pop the 16 bit value into a register like ax and then store the L.O. byte of that register into the byte variable. The second solution is to reserve an extra byte of padding after the byte variable to hold the whole word you will pop. Most programs use the former approach.

6.3.6 The LAHF and SAHF Instructions

The lahf (load ah from flags) and sahf (store ah into flags) instructions are archaic instructions included in the 80x86's instruction set to help improve compatibility with Intel's older 8080 µP chip. As such, these instructions have very little use in modern day 80x86 programs. The lahf instruction does not affect any of the flag bits. The sahf instruction, by its very nature, modifies the S, Z, A, P, and C bits in the processor status register. These instructions do not require any operands and you use them in the following manner:

```
sahf
lahf
```

Sahf only affects the L.O. eight bits of the flags register. Likewise, lahf only loads the L.O. eight bits of the flags register into the AH register. These instructions do not deal with the overflow, direction, interrupt disable, or trace flags. The fact that these instructions do not deal with the overflow flag is an important limitation.

Sahf has one major use. When using a floating point processor (8087, 80287, 80387, 80486, Pentium, etc.) you can use the sahf instruction to copy the floating point status register flags into the 80x86's flag register. You'll see this use in the chapter on floating point arithmetic (see "Floating Point Arithmetic" on page 771).

6.4 Conversions

The 80x86 instruction set provides several conversion instructions. They include movzx, movsx, cbw, cwd, cwde, cdq, bswap, and xlat. Most of these instructions sign or zero extend values, the last two convert between storage formats and translate values via a lookup table. These instructions take the general form:

```
movzx    dest, src    ;Dest must be twice the size of src.
movsx    dest, src    ;Dest must be twice the size of src.
cbw
cwd
cwde
cdq
bswap    reg32
xlat                  ;Special form allows an operand.
```

6.4.1 The MOVZX, MOVSX, CBW, CWD, CWDE, and CDQ Instructions

These instructions zero and sign extend values. The cbw and cwd instructions are available on all 80x86 processors. The movzx, movsx, cwde, and cdq instructions are available only on 80386 and later processors.

The cbw (convert byte to word) instruction sign extends the eight bit value in al to ax. That is, it copies bit seven of AL throughout bits 8-15 of ax. This instruction is especially important before executing an eight bit division (as you'll see in the section "Arithmetic Instructions" on page 255). This instruction requires no operands and you use it as follows:

```
cbw
```

The cwd (convert word to double word) instruction sign extends the 16 bit value in ax to 32 bits and places the result in dx:ax. It copies bit 15 of ax throughout the bits in dx. It is available on all 80x86 processors which explains why it doesn't sign extend the value into eax. Like the cbw instruction, this instruction is very important for division operations. Cwd requires no operands and you use it as follows

```
cwd
```

The cwde instruction sign extends the 16 bit value in ax to 32 bits and places the result in eax by copying bit 15 of ax throughout bits 16..31 of eax. This instruction is available only on the 80386 and later processors. As with cbw and cwd the instruction has no operands and you use it as follows:

```
cwde
```

The cdq instruction sign extends the 32 bit value in eax to 64 bits and places the result in edx:eax by copying bit 31 of eax throughout bits 0..31 of edx. This instruction is available only on the 80386 and later. You would normally use this instruction before a long division operation. As with cbw, cwd, and cwde the instruction has no operands and you use it as follows:

```
cdq
```

If you want to sign extend an eight bit value to 32 or 64 bits using these instructions, you could use sequences like the following:

```
; Sign extend al to dx:ax

        cbw
        cwd

; Sign extend al to eax

        cbw
        cwde

; Sign extend al to edx:eax

        cbw
        cwde
        cdq
```

You can also use the movsx for sign extensions from eight to sixteen or thirty-two bits.

The movsx instruction is a generalized form of the cbw, cwd, and cwde instructions. It will sign extend an eight bit value to a sixteen or thirty-two bits, or sign extend a sixteen bit value to a thirty-two bits. This instruction uses a mod-reg-r/m byte to specify the two operands. The allowable forms for this instruction are

```
movsx   reg16, mem8
movsx   reg16, reg8
movsx   reg32, mem8
movsx   reg32, reg8
movsx   reg32, mem16
movsx   reg32, reg16
```

Note that anything you can do with the cbw and cwde instructions, you can do with a movsx instruction:

```
movsx   ax, al      ;CBW
movsx   eax, ax     ;CWDE
movsx   eax, al     ;CBW followed by CWDE
```

However, the cbw and cwde instructions are shorter and sometimes faster. This instruction is available only on the 80386 and later processors. Note that there are not direct movsx equivalents for the cwd and cdq instructions.

The movzx instruction works just like the movsx instruction, except it extends unsigned values via zero extension rather than signed values through sign extension. The syntax is the same as for the movsx instructions except, of course, you use the movzx mnemonic rather than movsx.

Note that if you want to zero extend an eight bit register to 16 bits (e.g., al to ax) a simple mov instruction is faster and shorter than movzx. For example,

```
mov     bh, 0
```

is faster and shorter than

```
movzx   bx, bl
```

Of course, if you move the data to a different 16 bit register (e.g., movzx bx, al) the movzx instruction is better.

Like the movsx instruction, the movzx instruction is available only on 80386 and later processors. The sign and zero extension instructions do not affect any flags.

6.4.2 The BSWAP Instruction

The bswap instruction, available only on 80486 (yes, 486) and later processors, converts between 32 bit *little endian* and *big endian* values. This instruction accepts only a single 32 bit register operand. It swaps the first byte with the fourth and the second byte with the third. The syntax for the instruction is

```
bswap     reg32
```

where reg_{32} is an 80486 32 bit general purpose register.

The Intel processor families use a memory organization known as *little endian byte organization*. In little endian byte organization, the L.O. byte of a multi-byte sequence appears at the lowest address in memory. For example, bits zero through seven of a 32 bit value appear at the lowest address; bits eight through fifteen appear at the second address in memory; bits 16 through 23 appear in the third byte, and bits 24 through 31 appear in the fourth byte.

Another popular memory organization is *big endian*. In the big endian scheme, bits twenty-four through thirty-one appear in the first (lowest) address, bits sixteen through twenty-three appear in the second byte, bits eight through fifteen appear in the third byte, and bits zero through seven appear in the fourth byte. CPUs such as the Motorola 68000 family used by Apple in their Macintosh computer and many RISC chips employ the big endian scheme.

Normally, you wouldn't care about byte organization in memory since programs written for an Intel processor in assembly language do not run on a 68000 processor. However, it is very common to exchange data between machines with different byte organizations. Unfortunately, 16 and 32 bit values on big endian machines do not produce correct results when you use them on little endian machines. This is where the bswap instruction comes in. It lets you easily convert 32 bit big endian values to 32 bit little endian values.

One interesting use of the bswap instruction is to provide access to a second set of 16 bit general purpose registers. If you are using only 16 bit registers in your code, you can double the number of available registers by using the bswap instruction to exchange the data in a 16 bit register with the H.O. word of a thirty-two bit register. For example, you can keep two 16 bit values in eax and move the appropriate value into ax as follows:

```
< Some computations that leave a result in AX >
          bswap     eax
< Some additional computations involving AX >
          bswap     eax
< Some computations involving the original value in AX >
          bswap     eax
< Computations involving the 2nd copy of AX from above >
```

You can use this technique on the 80486 to obtain two copies of ax, bx, cx, dx, si, di, and bp. You must exercise extreme caution if you use this technique with the sp register.

Note: to convert 16 bit big endian values to 16 bit little endian values just use the 80x86 xchg instruction. For example, if ax contains a 16 bit big endian value, you can convert it to a 16 bit little endian value (or vice versa) using:

```
xchg      al, ah
```

The bswap instruction does not affect any flags in the 80x86 flags register.

6.4.3 The XLAT Instruction

The xlat instruction translates the value in the al register based on a lookup table in memory. It does the following:

```
temp := al+bx
al := ds:[temp]
```

that is, bx points at a table in the current data segment. Xlat replaces the value in al with the byte at the offset originally in al. If al contains four, xlat replaces the value in al with the fifth item (offset four) within the table pointed at by ds:bx. The xlat instruction takes the form:

```
xlat
```

Typically it has no operand. You can specify one but the assembler virtually ignores it. The only purpose for specifying an operand is so you can provide a segment override prefix:

```
xlat      es:Table
```

This tells the assembler to emit an es: segment prefix byte before the instruction. You must still load bx with the address of Table; the form above does not provide the address of Table to the instruction. Only the segment override prefix in the operand is significant.

The xlat instruction does not affect the 80x86's flags register.

6.5 Arithmetic Instructions

The 80x86 provides many arithmetic operations: addition, subtraction, negation, multiplication, division/modulo (remainder), and comparing two values. The instructions that handle these operations are add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg, and some miscellaneous conversion instructions: aaa, aad, aam, aas, daa, and das. The following sections describe these instructions in detail.

The generic forms for these instructions are

```
add    dest, src              dest := dest + src
adc    dest, src              dest := dest + src + C
SUB    dest, src              dest := dest - src
sbb    dest, src              dest := dest - src - C
mul    src                    acc := acc * src
imul   src                    acc := acc * src
imul   dest, src1, imm_src    dest := src1 * imm_src
imul   dest, imm_src          dest := dest * imm_src
imul   dest, src              dest := dest * src
div    src                    acc := xacc /-mod src
idiv   src                    acc := xacc /-mod src
cmp    dest, src              dest - src (and set flags)
neg    dest                   dest := - dest
inc    dest                   dest := dest + 1
dec    dest                   dest := dest - 1
xadd   dest, src              (see text)
cmpxchg operand1, operand2    (see text)
cmpxchg8ax, operand           (see text)
aaa                           (see text)
aad                           (see text)
aam                           (see text)
aas                           (see text)
daa                           (see text)
das                           (see text)
```

6.5.1 The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA

These instructions take the forms:

```
add     reg, reg
add     reg, mem
add     mem, reg
add     reg, immediate data
add     mem, immediate data
add     eax/ax/al, immediate data

adc forms are identical to ADD.

inc     reg
inc     mem
inc     reg16
xadd    mem, reg
xadd    reg, reg
aaa
daa
```

Note that the aaa and daa instructions use the implied addressing mode and allow no operands.

6.5.1.1 The ADD and ADC Instructions

The syntax of add and adc (add with carry) is similar to mov. Like mov, there are special forms for the ax/eax register that are more efficient. Unlike mov, you cannot add a value to a segment register with these instructions.

The add instruction adds the contents of the source operand to the destination operand. For example, add ax, bx adds bx to ax leaving the sum in the ax register. Add computes dest :=dest+source while adc computes dest :=dest+source+C where C represents the value in the carry flag. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.

Both instructions affect the flags identically. They set the flags as follows:

- The overflow flag denotes a signed arithmetic overflow.
- The carry flag denotes an unsigned arithmetic overflow.
- The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).
- The zero flag is set if the result of the addition is zero.
- The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.
- The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there are an even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote *even parity*). If there are an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote *odd parity*).

The add and adc instructions do not affect any other flags.

The add and adc instructions allow eight, sixteen, and (on the 80386 and later) thirty-two bit operands. Both source and destination operands must be the same size. See Chapter Nine if you want to add operands whose size is different.

Since there are no memory to memory additions, you must load memory operands into registers if you want to add two variables together. The following code examples demonstrate possible forms for the add instruction:

```
; J:= K + M

                mov     ax, K
                add     ax, M
                mov     J, ax
```

If you want to add several values together, you can easily compute the sum in a single register:

```
; J := K + M + N + P

                mov     ax, K
                add     ax, M
                add     ax, N
                add     ax, P
                mov     J, ax
```

If you want to reduce the number of hazards on an 80486 or Pentium processor, you can use code like the following:

```
                mov     bx, K
                mov     ax, M
                add     bx, N
                add     ax, P
                add     ax, bx
                mov     J, ax
```

One thing that beginning assembly language programmers often forget is that you can add a register to a memory location. Sometimes beginning programmers even believe that both operands have to be in registers, completely forgetting the lessons from Chapter Four. The 80x86 is a CISC processor that allows you to use memory addressing modes with various instructions like add. It is often more efficient to take advantages of the 80x86's memory addressing capabilities

```
; J := K + J

                mov     ax, K           ;This works because addition is
                add     J, ax           ; commutative!

; Often, beginners will code the above as one of the following two sequences.
; This is unnecessary!

                mov     ax, J           ;Really BAD way to compute
                mov     bx, K           ; J := J + K.
                add     ax, bx
                mov     J, ax

                mov     ax, J           ;Better, but still not a good way to
                add     ax, K           ; compute J := J + K
                mov     J, ax
```

Of course, if you want to add a constant to a memory location, you only need a single instruction. The 80x86 lets you directly add a constant to memory:

```
; J := J + 2

                add     J, 2
```

There are special forms of the add and adc instructions that add an immediate constant to the al, ax, or eax register. These forms are shorter than the standard add reg, immediate instruction. Other instructions also provide shorter forms when using these registers; therefore, you should try to keep computations in the accumulator registers (al, ax, and eax) as much as possible.

```
                add     bl, 2           ;Three bytes long
                add     al, 2           ;Two bytes long
                add     bx, 2           ;Four bytes long
                add     ax, 2           ;Three bytes long
                etc.
```

Another optimization concerns the use of small signed constants with the add and adc instructions. If a value is in the range -128,,+127, the add and adc instructions will sign extend an eight bit immediate constant to the necessary destination size (eight, sixteen, or thirty-two bits). Therefore, you should try to use small constants, if possible, with the add and adc instructions.

6.5.1.2 The INC Instruction

The inc (increment) instruction adds one to its operand. Except for the carry flag, inc sets the flags the same way as add operand, 1 would.

Note that there are two forms of inc for 16 or 32 bit registers. They are the inc reg and inc reg_{16} instructions. The inc reg and inc mem instructions are the same. This instruction consists of an opcode byte followed by a mod-reg-r/m byte (see Appendix D for details). The inc reg_{16} instruction has a single byte opcode. Therefore, it is shorter and usually faster.

The inc operand may be an eight bit, sixteen bit, or (on the 80386 and later) thirty-two bit register or memory location.

The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction. Indeed, the inc reg_{16} instruction is one byte long, so it turns out that *two* such instructions are shorter than the comparable add reg, 1 instruction; however, the two increment instructions will run slower on most modern members of the 80x86 family.

The inc instruction is very important because adding one to a register is a very common operation. Incrementing loop control variables or indices into an array is a very common operation, perfect for the inc instruction. The fact that inc does not affect the carry flag is very important. This allows you to increment array indices without affecting the result of a multiprecision arithmetic operation (see “Arithmetic and Logical Operations” on page 459 for more details about multiprecision arithmetic).

6.5.1.3 The XADD Instruction

Xadd (Exchange and Add) is another 80486 (and later) instruction. It does not appear on the 80386 and earlier processors. This instruction adds the source operand to the destination operand and stores the sum in the destination operand. However, just before storing the sum, it copies the original value of the destination operand into the source operand. The following algorithm describes this operation:

```
xadd dest, source

temp := dest
dest := dest + source
source := temp
```

The xadd sets the flags just as the add instruction would. The xadd instruction allows eight, sixteen, and thirty-two bit operands. Both source and destination operands must be the same size.

6.5.1.4 The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. Beyond this chapter, this text will not cover BCD or ASCII arithmetic since it is mainly for controller applications, not general purpose programming applications. BCD values are decimal integer coded in binary form with one decimal digit (0..9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero.

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add them as though they were binary numbers and then execute the daa instruction afterwards to correct the results. Likewise, you can use the aaa instruction to adjust the result of an ASCII addition after executing an add instruction. Please note that these two instructions assume that the add operands were proper decimal or ASCII values. If you add binary

(non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

The choice of the name "ASCII arithmetic" is unfortunate, since these values are not true ASCII characters. A name like "unpacked BCD" would be more appropriate. However, Intel uses the name ASCII, so this text will do so as well to avoid confusion. However, you will often hear the term "unpacked BCD" to describe this data type.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC[5] =1) ) then
                if (8088 or 8086)[6] then
                        al := al + 6
                else
                        ax := ax + 6
                endif
                ah := ah + 1
                AuxC := 1                       ;Set auxilliary carry
                Carry := 1                      ; and carry flags.
else
                AuxC := 0                       ;Clear auxilliary carry
                Carry := 0                      ; and carry flags.
endif
al := al and 0Fh
```

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers. This text will not deal with BCD or ASCII numeric strings, so you can safely ignore this instruction for now. Of course, you can use the aaa instruction any time you need to use the algorithm above, but that would probably be a rare situation.

The daa instruction functions like aaa except it handles packed BCD (binary code decimal) values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

```
if ( (AL and 0Fh) > 9 or (AuxC = 1)) then
                al := al + 6
                AuxC := 1                       ;Set Auxilliary carry.
endif
if ( (al > 9Fh) or (Carry = 1)) then
                al := al + 60h
                Carry := 1;                     ;Set carry flag.
endif
```

6.5.2 The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS

The sub (subtract), sbb (subtract with borrow), dec (decrement), aas (ASCII adjust for subtraction), and das (decimal adjust for subtraction) instructions work as you expect. Their syntax is very similar to that of the add instructions:

```
sub     reg, reg
sub     reg, mem
sub     mem, reg
sub     reg, immediate data
sub     mem, immediate data
sub     eax/ax/al, immediate data
```

5. AuxC denotes the *auxiliary carry* flag in the flags register.
6. The 8086/8088 work differently from the later processors, but for all valid operands all 80x86 processors produce correct results.

```
                sbb forms are identical to sub.

                dec     reg
                dec     mem
                dec     reg16
                aas
                das
```

The sub instruction computes the value dest := dest - src. The sbb instruction computes dest := dest - src - C. Note that subtraction is not commutative. If you want to compute the result for dest := src - dest you will need to use several instructions, assuming you need to preserve the source operand).

One last subject worth discussing is how the sub instruction affects the 80x86 flags register[7]. The sub, sbb, and dec instructions affect the flags as follows:

- They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.
- These instructions set the sign flag if the result is negative.
- These instructions set the overflow flag if signed overflow/underflow occurs.
- They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.
- They set the parity flag according to the number of one bits appearing in the result value.
- The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0..9) per byte. You would use this instruction after a sub or sbb instruction on the ASCII value. This instruction uses the following algorithm:

```
if ( (al and 0Fh) > 9 or AuxC = 1) then
                al := al - 6
                ah := ah - 1
                AuxC := 1               ;Set auxilliary carry
                Carry := 1              ; and carry flags.
else
                AuxC := 0               ;Clear Auxilliary carry
                Carry := 0              ; and carry flags.
endif
al := al and 0Fh
```

The das instruction handles the same operation for BCD values, it uses the following algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC = 1)) then
                al := al -6
                AuxC = 1
endif
if (al > 9Fh or Carry = 1) then
                al := al - 60h
                Carry := 1              ;Set the Carry flag.
endif
```

Since subtraction is not commutative, you cannot use the sub instruction as freely as the add instruction. The following examples demonstrate some of the problems you may encounter.

```
; J := K - J
                mov     ax, K           ;This is a nice try, but it computes
                sub     J, ax           ; J := J - K, subtraction isn't
                                        ; commutative!
```

7. The SBB instruction affects the flags in a similar fashion, just don't forget that SBB computes dest-source-C.

```
                mov     ax, K       ;Correct solution.
                sub     ax, J
                mov     J, ax

; J := J - (K + M) -- Don't forget this is equivalent to J := J - K - M

                mov     ax, K       ;Computes AX := K + M
                add     ax, M
                sub     J, ax       ;Computes J := J - (K + M)

                mov     ax, J       ;Another solution, though less
                sub     ax, K       ;Efficient
                sub     ax, M
                mov     J, ax
```

Note that the sub and sbb instructions, like add and adc, provide short forms to subtract a constant from an accumulator register (al, ax, or eax). For this reason, you should try to keep arithmetic operations in the accumulator registers as much as possible. The sub and sbb instructions also provide a shorter form when subtracting constants in the range -128..+127 from a memory location or register. The instruction will automatically sign extend an eight bit signed value to the necessary size before the subtraction occurs. See Appendix D for the details.

In practice, there really isn't a need for an instruction that subtracts a constant from a register or memory location – adding a negative value achieves the same result. Nevertheless, Intel provides a subtract immediate instruction.

After the execution of a sub instruction, the condition code bits (carry, sign, overflow, and zero) in the flags register contain values you can test to see if one of sub's operands is equal, not equal, less than, less than or equal, greater than, or greater than or equal to the other operand. See the cmp instruction for more details.

6.5.3 The CMP Instruction

The cmp (compare) instruction is identical to the sub instruction with one crucial difference – it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is

```
                cmp     dest, src
```

The specific forms are

```
                cmp     reg, reg
                cmp     reg, mem
                cmp     mem, reg
                cmp     reg, immediate data
                cmp     mem, immediate data
                cmp     eax/ax/al, immediate data
```

The cmp instruction updates the 80x86's flags according to the result of the subtraction operation (dest - src). You can test the result of the comparison by checking the appropriate flags in the flags register. For details on how this is done, see "The "Set on Condition" Instructions" on page 281 and "The Conditional Jump Instructions" on page 296.

Usually you'll want to execute a conditional jump instruction after a cmp instruction. This two step process, comparing two values and setting the flag bits then testing the flag bits with the conditional jump instructions, is a very efficient mechanism for making decisions in a program.

Probably the first place to start when exploring the cmp instruction is to take a look at exactly how the cmp instruction affects the flags. Consider the following cmp instruction:

```
                cmp     ax, bx
```

This instruction performs the computation ax-bx and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative. At first glance, you might think that this flag would be set if ax is less than bx but this isn't always the case. If ax=7FFFh and bx=-1 (0FFFFh) subtracting ax from bx produces 8000h, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn't contain the proper status. For unsigned operands, consider ax=0FFFFh and bx=1. Ax is greater than bx but their difference is 0FFFEh which is still negative. As it turns out, the sign flag and the overflow flag, taken together, can be used for comparing two signed values.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflow or underflow. As mentioned above, the sign flag and the overflow flag are both used when performing signed comparisons.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow. This occurs only when ax is less than bx where ax and bx are both unsigned values.

The cmp instruction also affects the parity and auxiliary carry flags, but you'll rarely test these two flags after a compare operation. Given that the cmp instruction sets the flags in this fashion, you can test the comparison of the two operands with the following flags:

```
cmp Oprnd1, Oprnd2
```

Table 27: Condition Code Settings After CMP

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Oprnd1 < Oprnd2 (C=1) Oprnd1 >= Oprnd2 (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:
If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Oprnd1 < Oprnd2 when using a signed comparison.
If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Oprnd1 >= Oprnd2 when using a signed comparison.

To understand why these flags are set in this manner, consider the following examples:

```
Oprnd1              minus       Oprnd2          S     O
------                          ------          -     -

0FFFF (-1)            -         0FFFE (-2)      0     0
08000                 -         00001           0     1
0FFFE (-2)            -         0FFFF (-1)      1     0
07FFF (32767)         -         0FFFF (-1)      1     1
```

Remember, the cmp operation is really a subtraction, therefore, the first example above computes (-1)-(-2) which is (+1). The result is positive and an overflow did not occur so both the S and O flags are zero. Since (S xor O) is zero, Oprnd1 is greater than or equal to Oprnd2.

In the second example, the cmp instruction would compute (-32768)-(+1) which is (-32769). Since a 16-bit signed integer cannot represent this value, the value wraps around to 7FFFh (+32767) and sets the overflow flag. Since the result is positive (at least within the confines of 16 bits) the sign flag is cleared. Since (S xor O) is one here, Oprnd1 is less than Oprnd2.

In the third example above, cmp computes (-2)-(-1) which produces (-1). No overflow occurred so the O flag is zero, the result is negative so the sign flag is one. Since (S xor O) is one, Oprnd1 is less than Oprnd2.

In the fourth (and final) example, cmp computes (+32767)-(-1). This produces (+32768), setting the overflow flag. Furthermore, the value wraps around to 8000h (-32768) so the sign flag is set as well. Since (S xor O) is zero, Oprnd1 is greater than or equal to Oprnd2.

6.5.4 The CMPXCHG, and CMPXCHG8B Instructions

The cmpxchg (compare and exchange) instruction is available only on the 80486 and later processors. It supports the following syntax:

```
cmpxchg     reg, reg
cmpxchg     mem, reg
```

The operands must be the same size (eight, sixteen, or thirty-two bits). This instruction also uses the accumulator register; it automatically chooses al, ax, or eax to match the size of the operands.

This instruction compares al, ax, or eax with the first operand and sets the zero flag if they are equal. If so, then cmpxchg copies the second operand into the first. If they are not equal, cmpxchg copies the first operand into the accumulator. The following algorithm describes this operation:

```
cmpxchg             operand1, operand2
if ({al/ax/eax} = operand1) then[8]
          zero := 1                         ;Set the zero flag
          operand1 := operand2
else
          zero := 0                         ;Clear the zero flag
          {al/ax/eax} := operand1
endif
```

Cmpxchg supports certain operating system data structures requiring atomic operations[9] and semaphores. Of course, if you can fit the above algorithm into your code, you can use the cmpxchg instruction as appropriate.

Note: unlike the cmp instruction, the cmpxchg instruction only affects the 80x86 zero flag. You cannot test other flags after cmpxchg as you could with the cmp instruction.

The Pentium processor supports a 64 bit compare and exchange instruction – cmpxchg8b. It uses the syntax:

```
cmpxchg8b   ax, mem64
```

This instruction compares the 64 bit value in edx:eax with the memory value. If they are equal, the Pentium stores ecx:ebx into the memory location, otherwise it loads edx:eax with the memory location. This instruction sets the zero flag according to the result. It does not affect any other flags.

6.5.5 The NEG Instruction

The neg (negate) instruction takes the two's complement of a byte or word. It takes a single (destination) operation and negates it. The syntax for this instruction is

```
neg       dest
```

It computes the following:

```
dest := 0 - dest
```

This effectively reverses the sign of the destination operand.

8. The choice of al, ax, or eax is made by the size of the operands. Both operands to cmpxchg must be the same size.

9. An atomic operation is one that the system cannot interrupt.

If the operand is zero, its sign does not change, although this clears the carry flag. Negating any other value sets the carry flag. Negating a byte containing -128, a word containing -32,768, or a double word containing -2,147,483,648 does not change the operand, but will set the overflow flag. Neg always updates the A, S, P, and Z flags as though you were using the sub instruction.

The allowable forms are:

```
neg     reg
neg     mem
```

The operands may be eight, sixteen, or (on the 80386 and later) thirty-two bit values.

Some examples:

```
; J := - J
                neg     J

; J := -K
                mov     ax, K
                neg     ax
                mov     J, ax
```

6.5.6 The Multiplication Instructions: MUL, IMUL, and AAM

The multiplication instructions provide you with your first taste of irregularity in the 8086's instruction set. Instructions like add, adc, sub, sbb, and many others in the 8086 instruction set use a mod-reg-r/m byte to support two operands. Unfortunately, there aren't enough bits in the 8086's opcode byte to support all instructions, so the 8086 uses the reg bits in the mod-reg-r/m byte as an opcode extension. For example, inc, dec, and neg do not require two operands, so the 80x86 CPUs use the reg bits as an extension to the eight bit opcode. This works great for single operand instructions, allowing Intel's designers to encode several instructions (eight, in fact) with a single opcode.

Unfortunately, the multiply instructions require special treatment and Intel's designers were still short on opcodes, so they designed the multiply instructions to use a single operand. The reg field contains an opcode extension rather than a register value. Of course, multiplication *is* a two operand function. The 8086 always assumes the accumulator (al, ax, or eax) is the destination operand. This irregularity makes using multiplication on the 8086 a little more difficult than other instructions because one operand has to be in the accumulator. Intel adopted this unorthogonal approach because they felt that programmers would use multiplication far less often than instructions like add and sub.

One problem with providing only a mod-reg-r/m form of the instruction is that you cannot multiply the accumulator by a constant; the mod-reg-r/m byte does not support the immediate addressing mode. Intel quickly discovered the need to support multiplication by a constant and provide some support for this in the 80286 processor[10]. This was especially important for multidimensional array access. By the time the 80386 rolled around, Intel generalized one form of the multiplication operation allowing standard mod-reg-r/m operands.

There are two forms of the multiply instruction: an unsigned multiplication (mul) and a signed multiplication (imul). Unlike addition and subtraction, you need separate instructions for these two operations.

The multiply instructions take the following forms:

10. On the original 8086 chip multiplication by a constant was always faster using shifts, additions, and subtractions. Perhaps Intel's designers didn't bother with multiplication by a constant for this reason. However, the 80286 multiply instruction was faster than the 8086 multiply instruction, so it was no longer true that multiplication was slower and the corresponding shift, add, and subtract instructions.

Unsigned Multiplication:

```
mul     reg
mul     mem
```

Signed (Integer) Multiplication:

```
imul    reg
imul    mem
imul    reg, reg, immediate          (2)
imul    reg, mem, immediate          (2)
imul    reg, immediate               (2)
imul    reg, reg                     (3)
imul    reg, mem                     (3)
```

BCD Multiplication Operations:

```
aam
```

```
2- Available on the 80286 and later, only.
3- Available on the 80386 and later, only.
```

As you can see, the multiply instructions are a real mess. Worse yet, you have to use an 80386 or later processor to get near full functionality. Finally, there are some restrictions on these instructions not obvious above. Alas, the only way to deal with these instructions is to memorize their operation.

Mul, available on all processors, multiplies unsigned eight, sixteen, or thirty-two bit operands. Note that when multiplying two n-bit values, the result may require as many as 2*n bits. Therefore, if the operand is an eight bit quantity, the result will require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits for the result.

The mul instruction, with an eight bit operand, multiplies the al register by the operand and stores the 16 bit result in ax. So

```
mul     operand8
```
or
```
imul    operand8
```

computes:

```
ax := al * operand8
```

"*" represents an unsigned multiplication for mul and a signed multiplication for imul.

If you specify a 16 bit operand, then mul and imul compute:

```
dx:ax := ax * operand16
```

"*" has the same meanings as above and dx:ax means that dx contains the H.O. word of the 32 bit result and ax contains the L.O. word of the 32 bit result.

If you specify a 32 bit operand, then mul and imul compute the following:

```
edx:eax := eax * operand32
```

"*" has the same meanings as above and edx:eax means that edx contains the H.O. double word of the 64 bit result and eax contains the L.O. double word of the 64 bit result.

If an 8x8, 16x16, or 32x32 bit product requires more than eight, sixteen, or thirty-two bits (respectively), the mul and imul instructions set the carry and overflow flags.

Mul and imul scramble the A, P, S, and Z flags. Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions.

Imul (integer multiplication) operates on signed operands. There are many different forms of this instruction as Intel attempted to generalize this instruction with successive processors. The previous paragraphs describe the first form of the imul instruction, with a single operand. The next three forms of the imul instruction are available only on the 80286 and later processors. They provide the ability to multiply a register by an immediate value. The last two forms, available only on 80386 and later processors, provide the ability to multiply an arbitrary register by another register or memory location. Expanded to show allowable operand sizes, they are

```
imul      operand1, operand2, immediate     ;General form

imul      reg16, reg16, immediate8
imul      reg16, reg16, immediate16
imul      reg16, mem16, immediate8
imul      reg16, mem16, immediate16
imul      reg16, immediate8
imul      reg16, immediate16
imul      reg32, reg32, immediate8          (3)
imul      reg32, reg32, immediate32         (3)
imul      reg32, mem32, immediate8          (3)
imul      reg32, mem32, immediate32         (3)
imul      reg32, immediate8                 (3)
imul      reg32, immediate32                (3)
```

3- Available on the 80386 and later, only.

The imul reg, immediate instructions are a special syntax the assembler provides. The encodings for these instructions are the same as imul reg, reg, immediate. The assembler simply supplies the same register value for both operands.

These instructions compute:

```
operand1 := operand2 * immediate
operand1 := operand1 * immediate
```

Besides the number of operands, there are several differences between these forms and the single operand mul/imul instructions:

- There isn't an 8x8 bit multiplication available (the $immediate_8$ operands simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).
- These instructions do not produce a 2*n bit result. That is, a 16x16 multiply produces a 16 bit result. Likewise, a 32x32 bit multiply produces a 32 bit result. These instructions set the carry and overflow flags if the result does not fit into the destination register.
- The 80286 version of imul allows an immediate operand, the standard mul/imul instructions do not.

The last two forms of the imul instruction are available only on 80386 and later processors. With the addition of these formats, the imul instruction is *almost* as general as the add instruction[11]:

```
imul      reg, reg
imul      reg, mem
```

These instructions compute

```
          reg := reg * reg
and       reg := reg * mem
```

Both operands must be the same size. Therefore, like the 80286 form of the imul instruction, you must test the carry or overflow flag to detect overflow. If overflow does occur, the CPU loses the H.O. bits of the result.

Important Note: Keep in mind that the zero flag contains an indeterminate result after executing a multiply instruction. You cannot test the zero flag to see if the result is zero after a multiplication. Likewise, these instructions scramble the sign flag. If you need to check these flags, compare the result to zero after testing the carry or overflow flags.

The aam (ASCII Adjust after Multiplication) instruction, like aaa and aas, lets you adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81, the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al:

11. There are still some restrictions on the size of the operands, e.g., no eight bit registers, you have to consider.

```
ah := ax div 10
al := ax mod 10
```

Unlike the other decimal/ASCII adjust instructions, assembly language programs regularly use aam since conversion between number bases uses this algorithm.

Note: the aam instruction consists of a two byte opcode, the second byte of which is the immediate constant 10. Assembly language programmers have discovered that if you substitute another immediate value for this constant, you can change the divisor in the above algorithm. This, however, is an undocumented feature. It works in all varieties of the processor Intel has produced to date, but there is no guarantee that Intel will support this in future processors. Of course, the 80286 and later processors let you multiply by a constant, so this trick is hardly necessary on modern systems.

There is no dam (decimal adjust for multiplication) instruction on the 80x86 processor.

Perhaps the most common use of the imul instruction is to compute offsets into multi-dimensional arrays. Indeed, this is probably the main reason Intel added the ability to multiply a register by a constant on the 80286 processor. In Chapter Four, this text used the standard 8086 mul instruction for array index computations. However, the extended syntax of the imul instruction makes it a much better choice as the following examples demonstrate:

```
MyArray         word    8 dup ( 7 dup ( 6 dup (?)))    ;8x7x6 array.
J               word    ?
K               word    ?
M               word    ?
                 .
                 .
                 .
; MyArray [J, K, M] := J + K - M

                mov     ax, J
                add     ax, K
                sub     ax, M

                mov     bx, J       ;Array index :=
                imul    bx, 7       ;          ((J*7 + K) * 6 + M) * 2
                add     bx, K
                imul    bx, 6
                add     bx, M
                add     bx, bx      ;BX := BX * 2

                mov     MyArray[bx], ax
```

Don't forget that the multiplication instructions are very slow; often an order of magnitude slower than an addition instruction. There are faster ways to multiply a value by a constant. See "Multiplying Without MUL and IMUL" on page 487 for all the details.

6.5.7 The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16 division or a 16/8 division. These instructions take the form:

```
div     reg         For unsigned division
div     mem

idiv    reg         For signed division
idiv    mem

aad                 ASCII adjust for division
```

The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in . With 32 bit operands (on the 80386 and later) div divides the 64 bit value in edx:eax by the operand leaving the quotient in eax and the remainder in edx.

You cannot, on the 80x86, simply divide one eight bit value by another. If the denominator is an eight bit value, the numerator must be a sixteen bit value. If you need to divide one unsigned eight bit value by another, you must zero extend the numerator to sixteen bits. You can accomplish this by loading the numerator into the al register and then moving zero into the ah register. Then you can divide ax by the denominator operand to produce the correct result. *Failing to zero extend al before executing div may cause the 80x86 to produce incorrect results!*

When you need to divide two 16 bit unsigned values, you must zero extend the ax register (which contains the numerator) into the dx register. Just load the immediate value zero into the dx register[12]. If you need to divide one 32 bit value by another, you must zero extend the eax register into edx (by loading a zero into edx) before the division.

When dealing with signed integer values, you will need to sign extend al to ax, ax to dx or eax into edx before executing idiv. To do so, use the cbw, cwd, cdq, or movsx instructions. If the H.O. byte or word does not already contain significant bits, then you must sign extend the value in the accumulator (al/ax/eax) before doing the idiv operation. Failure to do so may produce incorrect results.

There is one other catch to the 80x86's divide instructions: you can get a fatal error when using this instruction. First, of course, you can attempt to divide a value by zero. Furthermore, the quotient may be too large to fit into the eax, ax, or al register. For example, the 16/8 division "8000h / 2" produces the quotient 4000h with a remainder of zero. 4000h will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *int 0* trap. This usually means BIOS will print "division by zero" or "divide error" and abort your program. If this happens to you, chances are you didn't sign or zero extend your numerator before executing the division operation. Since this error will cause your program to crash, you should be very careful about the values you select when using division.

The auxiliary carry, carry, overflow, parity, sign, and zero flags are undefined after a division operation. If an overflow occurs (or you attempt a division by zero) then the 80x86 executes an INT 0 (interrupt zero).

Note that the 80286 and later processors do not provide special forms for idiv as they do for imul. Most programs use division far less often than they use multiplication, so Intel's designers did not bother creating special instructions for the divide operation. Note that there is no way to divide by an immediate value. You must load the immediate value into a register or a memory location and do the division through that register or memory location.

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation. It splits apart unpacked binary coded decimal values before an ASCII division operation. Although this text will not cover BCD arithmetic, the aad instruction is useful for other operations. The algorithm that describes this instruction is

```
al := ah*10 + al
ah := 0
```

This instruction is quite useful for converting strings of digits into integer values (see the questions at the end of this chapter).

The following examples show how to divide one sixteen bit value by another.

```
; J := K / M (unsigned)

                mov     ax, K       ;Get dividend
                mov     dx, 0       ;Zero extend unsigned value in AX to DX.

        < In practice, we should verify that M does not contain zero here >

                div     M
                mov     J, ax

; J := K / M (signed)
```

12. Or use the MOVZX instruction on the 80386 and later processors.

```
                mov     ax, K       ;Get dividend
                cwd                 ;Sign extend signed value in AX to DX.
        < In practice, we should verify that M does not contain zero here >
                idiv    M
                mov     J, ax
; J := (K*M)/P
                mov     ax, K       ;Note that the imul instruction produces
                imul    M           ; a 32 bit result in DX:AX, so we don't
                idiv    P           ; need to sign extend AX here.
                mov     J, ax       ;Hope and pray result fits in 16 bits!
```

6.6 Logical, Shift, Rotate and Bit Instructions

The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror, rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar. The 80386 and later processors provide an even richer set of operations. These are bt, bts, btr, btc, bsf, bsr, shld, shrd, and the conditional set instructions (set*cc*).

These instructions can manipulate bits, convert values, do logical operations, pack and unpack data, and do arithmetic operations. The following sections describe each of these instructions in detail.

6.6.1 The Logical Instructions: AND, OR, XOR, and NOT

The 80x86 logical instructions operate on a bit-by-bit basis. Both eight, sixteen, and thirty-two bit versions of each instruction exist. The and, not, or, and xor instructions do the following:

```
        and     dest, source                    ;dest := dest and source
        or      dest, source                    ;dest := dest or source
        xor     dest, source                    ;dest := dest xor source
        not     dest                            ;dest := not dest
```

The specific variations are

```
        and     reg, reg
        and     mem, reg
        and     reg, mem
        and     reg, immediate data
        and     mem, immediate data
        and     eax/ax/al, immediate data

        or uses the same formats as AND
        xor uses the same formats as AND

        not     register
        not     mem
```

Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

Testing the zero flag after these instructions is particularly useful. The and instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions (since this would produce a zero result); for example, if the source operand contained a

single one bit, then the zero flag will be set if the corresponding destination bit is zero, it will be one otherwise. The or instruction will only set the zero flag if both operands contain zero. The xor instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form

```
xor     reg16, reg16
```

is shorter than the comparable `mov reg,0` instruction.

Like the addition and subtraction instructions, the and, or, and xor instructions provide special forms involving the accumulator register and immediate data. These forms are shorter and sometimes faster than the general "register, immediate" forms. Although one does not normally think of operating on signed data with these instructions, the 80x86 does provide a special form of the "reg/mem, immediate" instructions that sign extend a value in the range -128..+127 to sixteen or thirty-two bits, as necessary.

The instruction's operands must all be the same size. On pre-80386 processors they can be eight or sixteen bits. On 80386 and later processors, they may be 32 bits long as well. These instructions compute the obvious bitwise logical operation on their operands, see Chapter One for details on these operations.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; see for more details. Likewise, you can use the or instruction to force certain bits to one in the destination operand; see "Masking Operations with the OR Instruction" on page 491 for the details. You can use these instructions, along with the shift and rotate instructions described next, to pack and unpack data. See "Packing and Unpacking Data Types" on page 491 for more details.

6.6.2 The Shift Instructions: SHL/SAL, SHR, SAR, SHLD, and SHRD

The 80x86 supports three different shift instructions (shl and sal are the same instruction): shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right). The 80386 and later processors provide two additional shifts: shld and shrd.

The shift instructions move bits around in a register or memory location. The general format for a shift instruction is

```
shl       dest, count
sal       dest, count
shr       dest, count
sar       dest, count
```

Dest is the value to shift and count specifies the number of bit positions to shift. For example, the shl instruction shifts the bits in the destination operand to the left the number of bit positions specified by the count operand. The shld and shrd instructions use the format:

```
shld      dest, source, count
shrd      dest, source, count
```

The specific forms for these instructions are

```
shl       reg, 1
shl       mem, 1
shl       reg, imm              (2)
shl       mem, imm              (2)
shl       reg, cl
shl       mem, cl

sal is a synonym for shl and uses the same formats.
shr uses the same formats as shl.
sar uses the same formats as shl.
```

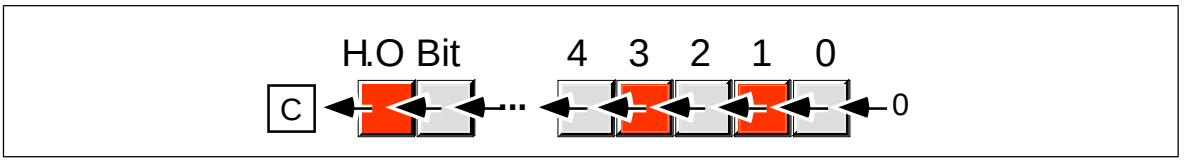

Figure 6.2 Shift Left Operation

```
shld      reg, reg, imm         (3)
shld      mem, reg, imm         (3)
shld      reg, reg, cl          (3)
shld      mem, reg, cl          (3)
```

shrd uses the same formats as shld.

```
2- This form is available on 80286 and later processors only.
3- This form is available on 80386 and later processors only.
```

For 8088 and 8086 CPUs, the number of bits to shift is either “1” or the value in cl. On 80286 and later processors you can use an eight bit immediate constant. Of course, the value in cl or the immediate constant should be less than or equal to the number of bits in the destination operand. It would be a waste of time to shift left al by nine bits (eight would produce the same result, as you will soon see). Algorithmically, you can think of the shift operations with a count other than one as follows:

```
for temp := 1 to count do
          shift dest, 1
```

There are minor differences in the way the shift instructions treat the overflow flag when the count is not one, but you can ignore this most of the time.

The shl, sal, shr, and sar instructions work on eight, sixteen, and thirty-two bit operands. The shld and shrd instructions work on 16 and 32 bit destination operands only.

6.6.2.1 SHL/SAL

The shl and sal mnemonics are synonyms. They represent the same instruction and use identical binary encodings. These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag (see Figure 6.2).

The shl/sal instruction sets the condition code bits as follows:

- If the shift count is zero, the shl instruction doesn’t affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- The overflow flag will contain one if the two H.O. bits were different prior to a single bit shift. The overflow flag is undefined if the shift count is not one.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

```
shl       ah, 4         ;This form requires an 80286 or later
or        al, ah        ;Merge in H.O. four bits.
```

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction). If the

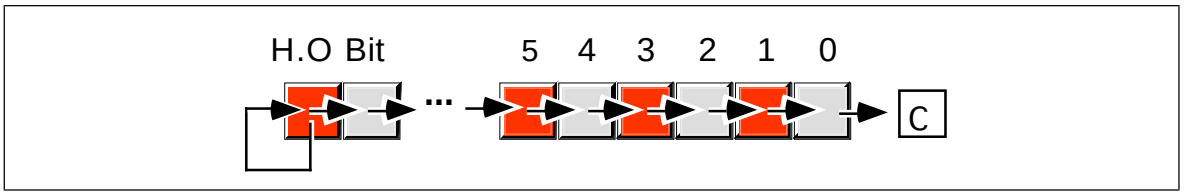

Figure 6.3 Arithmetic Shift Right Operation

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

```
shl     ah, 4       ;Move L.O. bits to H.O. position.
and     al, 0Fh     ;Clear H.O. four bits.
or      al, ah      ;Merge the bits.
```

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the shift left instruction for multiplication by powers of two:

```
shl     ax, 1       ;Equivalent to AX*2
shl     ax, 2       ;Equivalent to AX*4
shl     ax, 3       ;Equivalent to AX*8
shl     ax, 4       ;Equivalent to AX*16
shl     ax, 5       ;Equivlaent to AX*32
shl     ax, 6       ;Equivalent to AX*64
shl     ax, 7       ;Equivalent to AX*128
shl     ax, 8       ;Equivalent to AX*256
etc.
```

Note that shl ax, 8 is equivalent to the following two instructions:

```
mov     ah, al
mov     al, 0
```

The shl/sal instruction multiplies both signed and unsigned values by two for each shift. This instruction sets the carry flag if the result does not fit in the destination operand (i.e., unsigned overflow occurs). Likewise, this instruction sets the overflow flag if the signed result does not fit in the destination operation. This occurs when you shift a zero into the H.O. bit of a negative number or you shift a one into the H.O. bit of a non-negative number.

6.6.2.2 SAR

The sar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit (see Figure 6.3).

The sar instruction sets the flag bits as follows:

- If the shift count is zero, the sar instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- The overflow flag will contain zero if the shift count is one. Overflow can never occur with this instruction. However, if the count is not one, the value of the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The auxiliary carry flag is always undefined after the sar instruction.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

```
sar     ax, 1       ;Signed division by 2
sar     ax, 2       ;Signed division by 4
sar     ax, 3       ;Signed division by 8
sar     ax, 4       ;Signed division by 16
sar     ax, 5       ;Signed division by 32
sar     ax, 6       ;Signed division by 64
sar     ax, 7       ;Signed division by 128
sar     ax, 8       ;Signed division by 256
```

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity. The following examples demonstrate the difference:

```
mov     ax, -15
cwd
mov     bx, 2
idiv                ;Produces -7

mov     ax, -15
sar     ax, 1       ;Produces -8
```

Keep this in mind if you use sar for integer division operations.

The sar ax, 8 instruction effectively copies ah into al and then sign extends al into ax. This is because sar ax, 8 will shift ah down into al but leave a copy of ah's H.O. bit in all the bit positions of ah. Indeed, you can use the sar instruction on 80286 and later processors to sign extend one register into another. The following code sequences provide examples of this usage:

```
; Equivalent to CBW:

                mov     ah, al
                sar     ah, 7

; Equivalent to CWD:

                mov     dx, ax
                sar     dx, 15

; Equivalent to CDQ:

                mov     edx, eax
                sar     edx, 31
```

Of course it may seem silly to use two instructions where a single instruction might suffice; however, the cbw, cwd, and cdq instructions only sign extend al into ax, ax into dx:ax, and eax into edx:eax. Likewise, the movsx instruction copies its sign extended operand into a destination operand twice the size of the source operand. The sar instruction lets you sign extend one register into another register of the same size, with the second register containing the sign extension bits:

```
; Sign extend bx into cx:bx

                mov     cx, bx
                sar     cx, 15
```

6.6.2.3 SHR

The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit (see Figure 6.4).

The shr instruction sets the flag bits as follows:

- If the shift count is zero, the shr instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- If the shift count is one, the overflow flag will contain the value of the H.O. bit of the operand prior to the shift (i.e., this instruction sets the

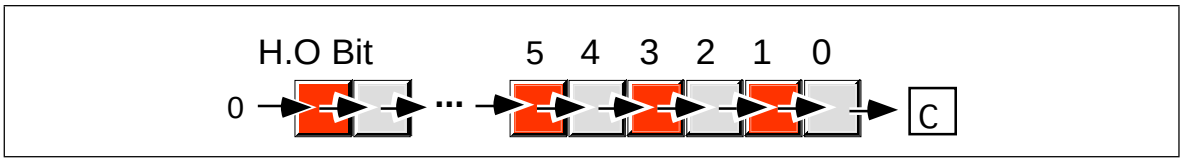

Figure 6.4 Shift Right Operation

overflow flag if the sign changes). However, if the count is not one, the value of the overflow flag is undefined.

- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result, which is always zero.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The auxiliary carry flag is always undefined after the shr instruction.

The shift right instruction is especially useful for unpacking data. For example, suppose you want to extract the two nibbles in the al register, leaving the H.O. nibble in ah and the L.O. nibble in al. You could use the following code to do this:

```
mov     ah, al      ;Get a copy of the H.O. nibble
shr     ah, 4       ;Move H.O. to L.O. and clear H.O. nibble
and     al, 0Fh     ;Remove H.O. nibble from al
```

Since shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

```
shr     ax, 1       ;Equivalent to AX/2
shr     ax, 2       ;Equivalent to AX/4
shr     ax, 3       ;Equivalent to AX/8
shr     ax, 4       ;Equivalent to AX/16
shr     ax, 5       ;Equivlaent to AX/32
shr     ax, 6       ;Equivalent to AX/64
shr     ax, 7       ;Equivalent to AX/128
shr     ax, 8       ;Equivalent to AX/256
etc.
```

Note that shr ax, 8 is equivalent to the following two instructions:

```
mov     al, ah
mov     ah, 0
```

Remember that division by two using shr only works for *unsigned* operands. If ax contains -1 and you execute shr ax, 1 the result in ax will be 32767 (7FFFh), not -1 or zero as you would expect. Use the sar instruction if you need to divide a signed integer by some power of two.

6.6.2.4 The SHLD and SHRD Instructions

The shld and shrd instructions provide double precision shift left and right operations, respectively. These instructions are available only on 80386 and later processors. Their generic forms are

```
shld    operand1, operand2, immediate
shld    operand1, operand2, cl
shrd    operand1, operand2, immediate
shrd    operand1, operand2, cl
```

$Operand_2$ must be a sixteen or thirty-two bit register. $Operand_1$ can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift.

The shld instruction shifts bits in $operand_1$ to the left. The H.O. bit shifts into the carry flag and the H.O. bit of $operand_2$ shifts into the L.O. bit of $operand_1$. Note that this instruc-

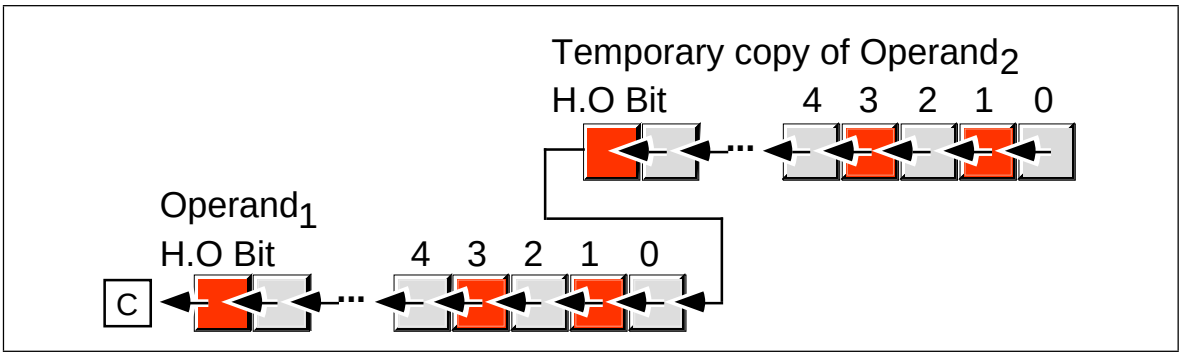

Figure 6.5 Double Precision Shift Left Operation

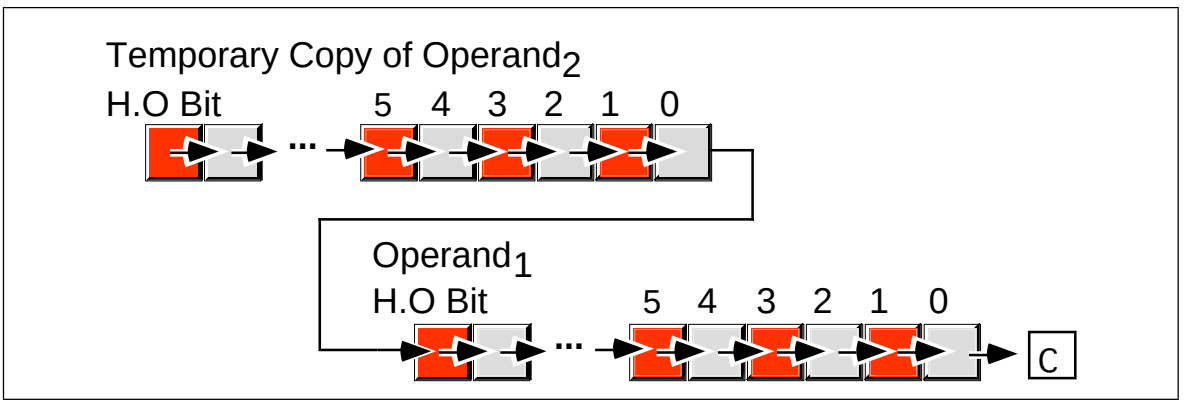

Figure 6.6 Double Precision Shift Right Operation

tion does not modify the value of operand$_2$, it uses a temporary copy of operand$_2$ during the shift. The immediate operand specifies the number of bits to shift. If the count is *n*, then shld shifts bit *n*-1 into the carry flag. It also shifts the H.O. *n* bits of operand$_2$ into the L.O. *n* bits of operand$_1$. Pictorially, the shld instruction appears in Figure 6.5.

The shld instruction sets the flag bits as follows:

- If the shift count is zero, the shld instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand$_1$.
- If the shift count is one, the overflow flag will contain one if the sign bit of operand$_1$ changes during the shift. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

The shld instruction is useful for packing data from many different sources. For example, suppose you want to create a word by merging the H.O. nibbles of four other words. You could do this with the following code:

```
mov     ax, Value4   ;Get H.O. nibble
shld    bx, ax, 4    ;Copy H.O. bits of AX to BX.
mov     ax, Value3   ;Get nibble #2.
shld    bx, ax, 4    ;Merge into bx.
mov     ax, Value2   ;Get nibble #1.
shld    bx, ax, 4    ;Merge into bx.
mov     ax, Value1   ;Get L.O. nibble
shld    bx, ax, 4    ;BX now contains all four nibbles.
```

The shrd instruction is similar to shld except, of course, it shifts its bits right rather than left. To get a clear picture of the shrd instruction, consider Figure 6.6.

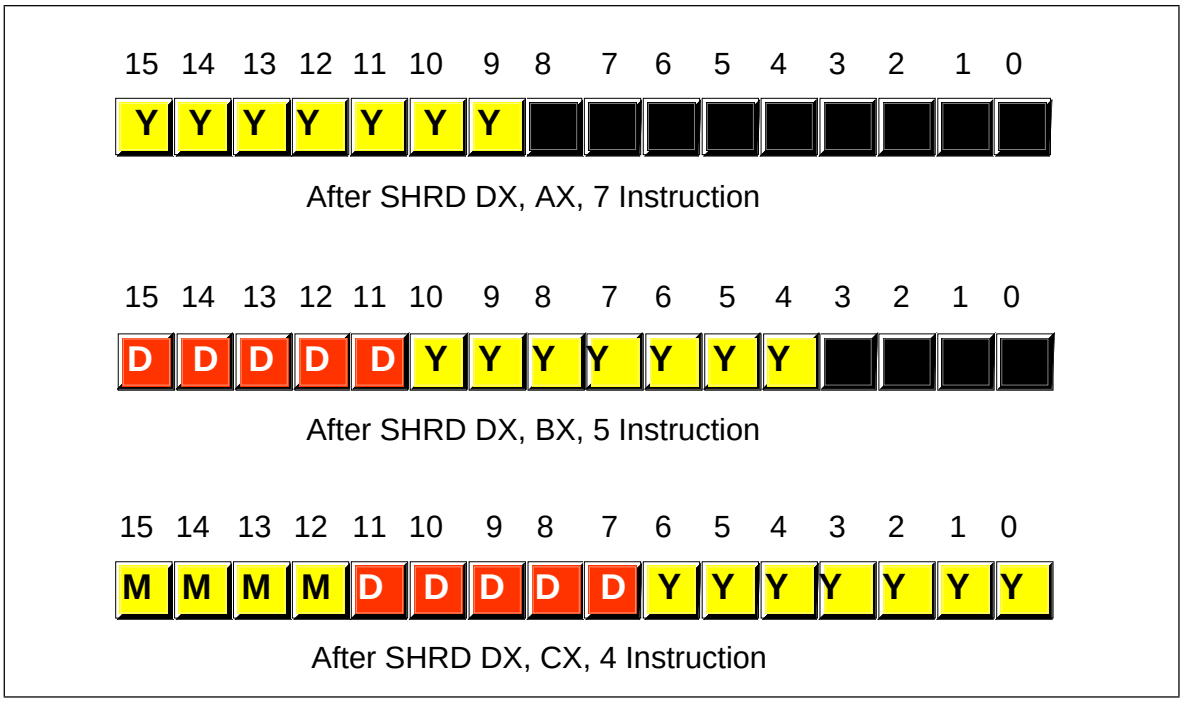

Figure 6.7 Packing Data with an SHRD Instruction

The shrd instruction sets the flag bits as follows:

- If the shift count is zero, the shrd instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand$_1$.
- If the shift count is one, the overflow flag will contain one if the H.O. bit of operand$_1$ changes. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

Quite frankly, these two instructions would probably be slightly more useful if Operand$_2$ could be a memory location. Intel designed these instructions to allow fast multiprecision (64 bits, or more) shifts. For more information on such usage, see "Extended Precision Shift Operations" on page 482.

The shrd instruction is marginally more useful than shld for packing data. For example, suppose that ax contains a value in the range 0..99 representing a year (1900..1999), bx contains a value in the range 1..31 representing a day, and cx contains a value in the range 1..12 representing a month (see "Bit Fields and Packed Data" on page 28). You can easily use the shrd instruction to pack this data into dx as follows:

```
shrd    dx, ax, 7
shrd    dx, bx, 5
shrd    dx, cx, 4
```

See Figure 6.7 for a blow-by-blow example.

6.6.3 The Rotate Instructions: RCL, RCR, ROL, and ROR

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the operand. They include rcl (rotate through carry left), rcr (rotate through carry right), rol (rotate left), and ror (rotate right). These instructions all take the forms:

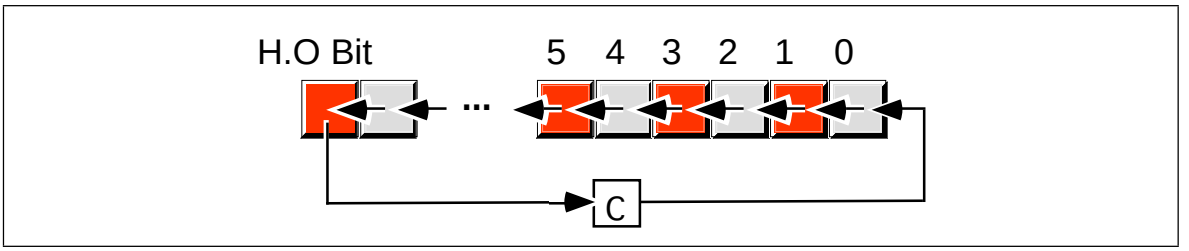

Figure 6.8 Rotate Through Carry Left Operation

```
rcl       dest, count
rol       dest, count
rcr       dest, count
ror       dest, count
```

The specific forms are

```
rcl       reg, 1
rcl       mem, 1
rcl       reg, imm    (2)
rcl       mem, imm    (2)
rcl       reg, cl
rcl       mem, cl

rol uses the same formats as rcl.
rcr uses the same formats as rcl.
ror uses the same formats as rcl.

2- This form is avialable on 80286 and later processors only.
```

6.6.3.1 RCL

The rcl (rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right (see Figure 6.8).

Note that if you rotate through carry an object *n+1* times, where *n* is the number of bits in the object, you wind up with your original value. Keep in mind, however, that some flags may contain different values after *n+1* rcl operations.

The rcl instruction sets the flag bits as follows:

- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
- The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

Important warning: unlike the shift instructions, the rotate instructions do not affect the sign, zero, parity, or auxiliary carry flags. This lack of orthogonality can cause you lots of grief if you forget it and attempt to test these flags after an rcl operation. If you need to test one of these flags after an rcl operation, test the carry and overflow flags first (if necessary) then compare the result to zero to set the other flags.

6.6.3.2 RCR

The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit (see Figure 6.9).

This instruction sets the flags in a manner analogous to rcl:

- The carry flag contains the last bit shifted out of the L.O. bit of the operand.

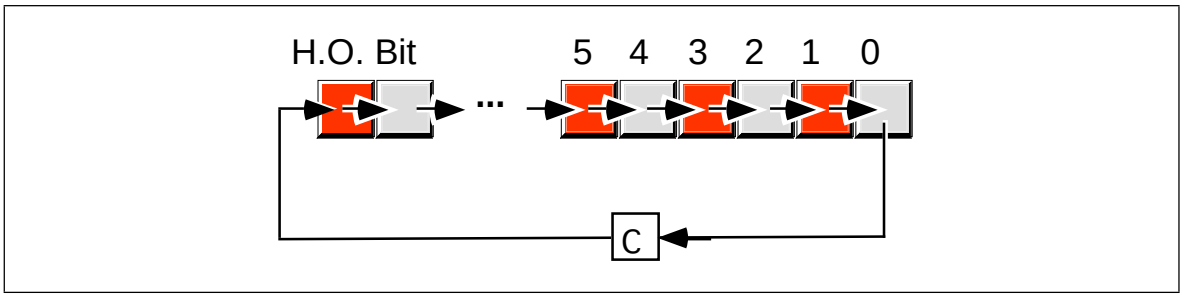

Figure 6.9 Rotate Through Carry Right Operation

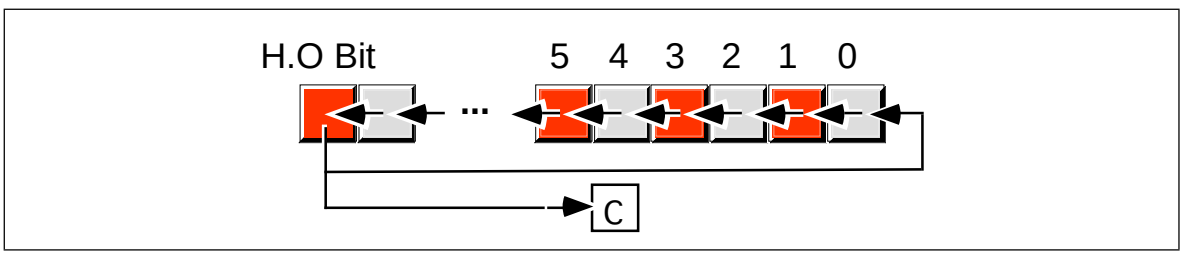

Figure 6.10 Rotate Left Operation

- If the shift count is one, then rcr sets the overflow flag if the sign changes (meaning the values of the H.O. bit and carry flag were not the same before the execution of the instruction). However, if the count is not one, the value of the overflow flag is undefined.
- The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

Keep in mind the warning given for rcl above.

6.6.3.3 ROL

The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit, rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag (see Figure 6.10).

The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction. **Don't forget the warning about the flags!**

Like shl, the rol instruction is often useful for packing and unpacking data. For example, suppose you want to extract bits 10..14 in ax and leave these bits in bits 0..4. The following code sequences will both accomplish this:

```
shr     ax, 10
and     ax, 1Fh

rol     ax, 6
and     ax, 1Fh
```

6.6.3.4 ROR

The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit (see Figure 6.11).

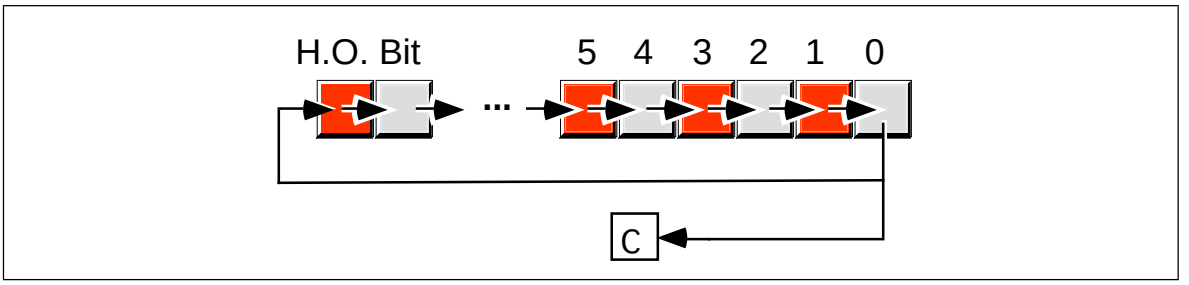

Figure 6.11 Rotate Right Operation

The ror instruction sets the flags identically to rcr. Other than the source of the bit shifted into the H.O. bit, this instruction behaves exactly like the rcr instruction. **Don't forget the warning about the flags!**

6.6.4 The Bit Operations

Bit twiddling is one of those operations easier done in assembly language than other languages. And no wonder. Most high-level languages shield you from the machine representation of the underlying data types[13]. Instructions like and, or, xor, not, and the shifts and rotates make it possible to test, set, clear, invert, and align bit fields within strings of bits. Even the C++ programming language, famous for its bit manipulation operators, doesn't provide the bit manipulation capabilities of assembly language.

The 80x86 family, particularly the 80386 and later processors, go much farther, though. Besides the standard logical, shift, and rotate instructions, there are instructions to test bits within an operand, to test and set, clear, or invert specific bits in an operand, and to search for set bits. These instructions are

```
test      dest, source
bt        source, index
btc       source, index
btr       source, index
bts       source, index
bsf       dest, source
bsr       dest, source
```

The specific forms are

```
test      reg, reg
test      reg, mem
test      mem, reg                        (*)
test      reg, imm
test      mem, imm
test      eax/ax/al, imm

bt        reg, reg                        (3)
bt        mem, reg                        (3)
bt        reg, imm                        (3)
bt        mem, imm                        (3)

btc uses the same formats as bt.          (3)
btr uses the same formats as bt.          (3)
bts uses the same formats as bt.          (3)

bsf       reg, reg                        (3)
bsr       reg, mem                        (3)

bsr uses the same formats as bsf.         (3)

3- This instruction is only available on 80386 and later processors.
*- This is the same instruction as test reg,mem
```

Note that the bt, btc, btr, bts, bsf, and bsr require 16 or 32 bit operands.

13. Indeed, this is one of the purposes of high level languages, to hide such low-level details.

The bit operations are useful when implementing (monochrome) bit mapped graphic primitive functions and when implementing a set data type using bit maps.

6.6.4.1 TEST

The test instruction logically ands its two operands and sets the flags but does not save the result. Test and and share the same relationship as cmp and sub. Typically, you would use this instruction to see if a bit contains one. Consider the following instruction:

```
test    al, 1
```

This instruction logically ands al with the value one. If bit zero of al contains a one, the result is non-zero and the 80x86 clears the zero flag. If bit zero of al contains zero, then the result is zero and the test operation sets the zero flag. You can test the zero flag after this instruction to decide whether al contained zero or one in bit zero.

The test instruction can also check to see if one or more bits in a register or memory location are non-zero. Consider the following instruction:

```
test    dx, 105h
```

This instruction logically ands dx with the value 105h. This will produce a non-zero result (and, therefore, clear the zero flag) if at least one of bits zero, two, or eight contain a one. They must all be zero to set the zero flag.

The test instruction sets the flags identically to the and instruction:

- It clears the carry flag.
- It clears the overflow flag.
- It sets the zero flag if the result is zero, they clear it otherwise.
- It copies the H.O. bit of the result into the sign flag.
- It sets the parity flag according to the parity (number of one bits) in the L.O. byte of the result.
- It scrambles the auxiliary carry flag.

6.6.4.2 The Bit Test Instructions: BT, BTS, BTR, and BTC

On an 80386 or later processor, you can use the bt instruction (*bit test*) to test a single bit. Its second operand specifies the *bit index* into the first operand. Bt copies the addressed bit into the carry flag. For example, the instruction

```
bt      ax, 12
```

copies bit twelve of ax into the carry flag.

The bt/bts/btr/btc instructions only deal with 16 or 32 bit operands. This is not a limitation of the instruction. After all, if you want to test bit three of the al register, you can just as easily test bit three of the ax register. On the other hand, if the index is larger than the size of a register operand, the result is undefined.

If the first operand is a memory location, the bt instruction tests the bit at the given offset in memory, regardless the value of the index. For example, if bx contains 65 then

```
bt      TestMe, bx
```

will copy bit one of location TestMe+8 into the carry flag. Once again, the size of the operand does not matter. For all intents and purposes, the memory operand is a byte and you can test any bit after that byte with an appropriate index. The actual bit bt tests is at bit position *index mod 8* and at memory offset *effective address + index/8*.

The bts, btr, and btc instructions also copy the addressed bit into the carry flag. However, these instructions also set, reset (clear), or complement (invert) the bit in the first operand after copying it to the carry flag. This provides *test and set*, *test and clear*, and *test and invert* operations necessary for some concurrent algorithms.

The bt, bts, btr, and btc instructions do not affect any flags other than the carry flag.

6.6.4.3 Bit Scanning: BSF and BSR

The bsf (Bit Scan Forward) and bsr (Bit Scan Reverse) instructions search for the first or last set bit in a 16 or 32 bit quantity. The general form of these instructions is

```
bsf     dest, source
bsr     dest, source
```

Bsf locates the first set bit in the source operand, searching from bit zero though the H.O. bit. Bsr locates the first set bit searching from the H.O. bit down to the L.O. bit. If these instructions locate a one, they clear the zero flag and store the bit index (0..31) into the destination operand. If the source operand is zero, these instructions set the zero flag and store an indeterminate value into the destination operand[14].

To scan for the first bit containing zero (rather than one), make a copy of the source operand and invert it (using not), then execute bsf or bsr on the inverted value. The zero flag would be set after this operation if there were no zero bits in the original source value, otherwise the destination operation will contain the position of the first bit containing zero.

6.6.5 The "Set on Condition" Instructions

The *set on condition* (or set*cc*) instructions set a single byte operand (register or memory location) to zero or one depending on the values in the flags register. The general formats for the set*cc* instructions are

```
setcc   reg8
setcc   mem8
```

Set*cc* represents a mnemonic appearing in the following tables. These instructions store a zero into the corresponding operand if the condition is false, they store a one into the eight bit operand if the condition is true.

Table 28: SETcc Instructions That Test Flags

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

14. On many of the processors, if the source operand is zero the CPU will leave the destination operand unchanged. However, certain versions of the 80486 do scramble the destination operand, so you shouldn't count on it being unchanged if the source operand is zero.

The set*cc* instructions above simply test the flags without any other meaning attached to the operation. You could, for example, use setc to check the carry flag after a shift, rotate, bit test, or arithmetic operation. Likewise, you could use setnz instruction after a test instruction to check the result.

The cmp instruction works synergistically with the set*cc* instructions. Immediately after a cmp operation the processor flags provide information concerning the relative values of those operands. They allow you to see if one operand is less than, equal to, greater than, or any combination of these.

There are two groups of set*cc* instructions that are very useful after a cmp operation. The first group deals with the result of an *unsigned* comparison, the second group deals with the result of a *signed* comparison.

Table 29: SETcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The corresponding table for signed comparisons is

Table 30: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw or Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE
SETNGE	Set if not greater or equal (not >=)	Sign ≠ Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The set*cc* instructions are particularly valuable because they can convert the result of a comparison to a boolean value (true/false or 0/1). This is especially important when

translating statements from a high level language like Pascal or C++ into assembly language. The following example shows how to use these instructions in this manner:

```
; Bool := A <= B

                mov     ax, A       ;Assume A and B are signed integers.
                cmp     ax, B
                setle   Bool        ;Bool needs to be a byte variable.
```

Since the set*cc* instructions always produce zero or one, you can use the results with the logical and and or instructions to compute complex boolean values:

```
; Bool := ((A <= B) and (D = E)) or (F <> G)

                mov     ax, A
                cmp     ax, B
                setle   bl
                mov     ax, D
                cmp     ax, E
                sete    bh
                and     bl, bh
                mov     ax, F
                cmp     ax, G
                setne   bh
                or      bl, bh
                mov     Bool, bh
```

For more examples, see “Logical (Boolean) Expressions” on page 467.

The set*cc* instructions always produce an eight bit result since a byte is the smallest operand the 80x86 will operate on. However, you can easily use the shift and rotate instructions to pack eight boolean values in a single byte. The following instructions compare eight different values with zero and copy the “zero flag” from each comparison into corresponding bits of al:

```
                cmp     Val7, 0
                setne   al          ;Put first value in bit #0
                cmp     Val6, 0     ;Test the value for bit #6
                setne   ah          ;Copy zero flag into ah register.
                shr     ah, 1       ;Copy zero flag into carry.
                rcl     al, 1       ;Shift carry into result byte.
                cmp     Val5, 0     ;Test the value for bit #5
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val4, 0     ;Test the value for bit #4
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val3, 0     ;Test the value for bit #3
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val2, 0     ;Test the value for bit #2
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val1, 0     ;Test the value for bit #1
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val0, 0     ;Test the value for bit #0
                setne   ah
                shr     ah, 1
                rcl     al, 1

; Now AL contains the zero flags from the eight comparisons.
```

6.7 I/O Instructions

The 80x86 supports two I/O instructions: in and out[15]. They take the forms:

```
in      eax/ax/al, port
in      eax/ax/al, dx
out     port, eax/ax/al
out     dx, eax/ax/al
```

port is a value between 0 and 255.

The 80x86 supports up to 65,536 different I/O ports (requiring a 16 bit I/O address). The port value above, however, is a single byte value. Therefore, you can only directly address the first 256 I/O ports in the 80x86's I/O address space. To address all 65,536 different I/O ports, you must load the address of the desired port (assuming it's above 255) into the dx register and access the port indirectly. The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Please realize that there is nothing magical about the 80x86's in and out instructions. They're simply another form of the mov instruction that accesses a different memory space (the I/O address space) rather than the 80x86's normal 1 Mbyte memory address space.

The in and out instructions do not affect any 80x86 flags.

Examples of the 80x86 I/O instructions:

```
in      al, 60h     ;Read keyboard port

mov     dx, 378h    ;Point at LPT1: data port
in      al, dx      ;Read data from printer port.
inc     ax          ;Bump the ASCII code by one.
out     dx, al      ;Write data in AL to printer port.
```

6.8 String Instructions

The 80x86 supports twelve string instructions:

- movs (move string)
- lods (load string element into the accumulator)
- stos (store accumulator into string element)
- scas (Scan string and check for match against the value in the accumulator)
- cmps (compare two strings)
- ins (input a string from an I/O port)
- outs (output a string to an I/O port
- rep (repeat a string operation)
- repz (repeat while zero)
- repe (repeat while equal)
- repnz (repeat while not zero)
- repne (repeat while not equal)

You can use the movs, stos, scas, cmps, ins and outs instructions to manipulate a single element (byte, word, or double word) in a string, or to process an entire string. Generally, you would only use the lods instruction to manipulate a single item at a time.

These instructions can operate on strings of bytes, words, or double words. To specify the object size, simply append a *b*, *w*, or *d* to the end of the instruction's mnemonic, i.e., lodsb, movsw, cmpsd, etc. Of course, the double word forms are only available on 80386 and later processors.

15. Actually, the 80286 and later processors support four I/O instructions, you'll get a chance to see the other two in the next section.

The movs and cmps instructions assume that ds:si contains the segmented address of a source string and that es:di contains the segmented address of a destination string. The lods instruction assumes that ds:si points at a source string, the accumulator (al/ax/eax) is the destination location. The scas and stos instructions assume that es:di points at a destination string and the accumulator contains the source value.

The movs instruction moves one string element (byte, word, or dword) from memory location ds:si to es:di. After moving the data, the instruction increments or decrements si and di by one, two, or four if processing bytes, words, or dwords, respectively. The CPU increments these registers if the direction flag is clear, the CPU decrements them if the direction flag is set.

The movs instruction can move blocks of data around in memory. You can use it to move strings, arrays, and other multi-byte data structures.

```
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;
```

Note: *size* is one for bytes, two for words, and four for dwords.

The cmps instruction compares the byte, word, or dword at location ds:si to es:di and sets the processor flags accordingly. After the comparison, cmps increments or decrements si and di by one, two, or four depending on the size of the instruction and the status of the direction flag in the flags register.

```
cmps{b,w,d}:    cmp ds:[si], es:[di]
                if direction_flag = 0 then
                        si := si + size;
                        di := di + size;
                else
                        si := si - size;
                        di := di - size;
                endif;
```

The lods instruction moves the byte, word, or dword at ds:si into the al, ax, or eax register. It then increments or decrements the si register by one, two, or four depending on the instruction size and the value of the direction flag. The lods instruction is useful for fetching a sequence of bytes, words, or double words from an array, performing some operation(s) on those values and then processing the next element from the string.

```
lods{b,w,d}:    eax/ax/al := ds:[si]
                if direction_flag = 0 then
                        si := si + size;
                else
                        si := si - size;
                endif;
```

The stos instruction stores al, ax, or eax at the address specified by es:di. Again, di is incremented or decremented according to the size of the instruction and the value of the direction flag. The stos instruction has several uses. Paired with the lods instruction above, you can load (via lods), manipulate, and store string elements. By itself, the stos instruction can quickly store a single value throughout a multi-byte data structure.

```
stos{b,w,d}:    es:[di] := eax/ax/al
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The scas instruction compares al, ax or eax against the value at location es:di and then adjusts di accordingly. This instruction sets the flags in the processor status register just

like the cmp and cmps instructions. The scas instruction is great for searching for a particular value throughout some multi-byte data structure.

```
scas{b,w,d}:    cmp eax/ax/al, es:[di]
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The ins instruction inputs a byte, word, or double word from the I/O port specified in the dx register. It then stores the input value at memory location es:di and increments or decrements di appropriately. This instruction is available only on 80286 and later processors.

```
ins{b,w,d}:     es:[di] := port(dx)
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The outs instruction fetches the byte, word, or double word at address ds:si, increments or decrements si accordingly, and then outputs the value to the port specified in the dx register.

```
outs{b,w,d}:    port(dx) := ds:[si]
                if direction_flag = 0 then
                        si := si + size;
                else
                        si := si - size;
                endif;
```

As explained here, the string instructions are useful, but it gets even better! When combined with the rep, repz, repe, repnz, and repne prefixes, a single string instruction can process an entire string. For more information on these prefixes see the chapter on strings.

6.9 Program Flow Control Instructions

The instructions discussed thus far execute sequentially; that is, the CPU executes each instruction in the sequence it appears in your program. To write real programs requires several control structures, not just the sequence. Examples include the if statement, loops, and subroutine invocation (a call). Since compilers reduce all other languages to assembly language, it should come as no surprise that assembly language supports the instructions necessary to implement these control structures. 80x86 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions. The following sections describe these instructions:

6.9.1 Unconditional Jumps

The jmp (jump) instruction unconditionally transfers control to another point in the program. There are six forms of this instruction: an intersegment/direct jump, two intrasegment/direct jumps, an intersegment/indirect jump, and two intrasegment/indirect jumps. Intrasegment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

These instructions generally use the same syntax, it is

```
jmp     target
```

The assembler differentiates them by their operands:

```
jmp     disp8       ;direct intrasegment, 8 bit displacement.
jmp     disp16      ;direct intrasegment, 16 bit displacement.
jmp     adrs32      ;direct intersegment, 32 bit segmented address.
```

```
        jmp     mem16       ;indirect intrasegment, 16 bit memory operand.
        jmp     reg16       ;register indirect intrasegment.
        jmp     mem32       ;indirect intersegment, 32 bit memory operand.
```

Intersegment is a synonym for *far*, intrasegment is a synonym for *near*.

The two direct intrasegment jumps differ only in their length. The first form consists of an opcode and a single byte displacement. The CPU sign extends this displacement to 16 bits and adds it to the ip register. This instruction can branch to a location -128..+127 from the beginning of the next instruction following it (i.e., -126..+129 bytes around the current instruction).

The second form of the intrasegment jump is three bytes long with a two byte displacement. This instruction allows an effective range of -32,768..+32,767 bytes and can transfer control to anywhere in the current code segment. The CPU simply adds the two byte displacement to the ip register.

These first two jumps use a *relative* addressing scheme. The offset encoded as part of the opcode byte is *not* the target address in the current code segment, but the distance to the target address. Fortunately, MASM will compute the distance for you automatically, so you do not have to compute this displacement value yourself. In many respects, these instructions are really nothing more than add ip, disp instructions.

The direct intersegment jump is five bytes long, the last four bytes containing a segmented address (the offset in the second and third bytes, the segment in the fourth and fifth bytes). This instruction copies the offset into the ip register and the segment into the cs register. Execution of the next instruction continues at the new address in cs:ip. Unlike the previous two jumps, the address following the opcode is the absolute memory address of the target instruction; this version does not use relative addressing. This instruction loads cs:ip with a 32 bit immediate value.

For the three direct jumps described above, you normally specify the target address using a *statement label*. A statement label is usually an identifier followed by a colon, usually on the same line as an executable machine instruction. The assembler determines the offset of the statement after the label and automatically computes the distance from the jump instruction to the statement label. Therefore, you do not have to worry about computing displacements manually. For example, the following short little loop continuously reads the parallel printer data port and inverts the L.O. bit. This produces a *square wave* electrical signal on one of the printer port output lines:

```
                mov     dx, 378h    ;Parallel printer port address.
LoopForever:    in      al, dx      ;Read character from input port.
                xor     al, 1       ;Invert the L.O. bit.
                out     dx, al      ;Output data back to port.
                jmp     LoopForever ;Repeat forever.
```

The fourth form of the unconditional jump instruction is the indirect intrasegment jump instruction. It requires a 16 bit memory operand. This form transfers control to the address within the offset given by the two bytes of the memory operand. For example,

```
WordVar         word    TargetAddress
                  .
                  .
                  .
                jmp     WordVar
```

transfers control to the address specified by the value in the 16 bit memory location WordVar. This does *not* jump to the statement at address WordVar, it jumps to the statement at the address held in the WordVar variable. Note that this form of the jmp instruction is roughly equivalent to:

```
                mov     ip, WordVar
```

Although the example above uses a single word variable containing the indirect address, you can use *any* valid memory address mode, not just the displacement only addressing mode. You can use memory indirect addressing modes like the following:

```
                jmp     DispOnly    ;Word variable
```

```
                jmp     Disp[bx]    ;Disp is an array of words
                jmp     Disp[bx][si]
                jmp     [bx][16]
                etc.
```

Consider the indexed addressing mode above for a moment (disp[bx]). This addressing mode fetches the word from location disp+bx and copies this value to the ip register; this lets you create an array of pointers and jump to a specified pointer using an array index. Consider the following example:

```
AdrsArray       word    stmt1, stmt2, stmt3, stmt4
                 .
                 .
                 .
                mov     bx, I       ;I is in the range 0..3
                add     bx, bx      ;Index into an array of words.
                jmp     AdrsArray[bx];Jump to stmt1, stmt2, etc., depending
                                    ; on the value of I.
```

The important thing to remember is that the *near indirect* jump fetches a word from memory and copies it into the ip register; it does *not* jump to the memory location specified, it jumps indirectly through the 16 bit pointer at the specified memory location.

The fifth jmp instruction transfers control to the offset given in a 16 bit general purpose register. Note that you can use *any* general purpose register, not just bx, si, di, or bp. An instruction of the form

```
                jmp     ax
```

is roughly equivalent to

```
                mov     ip, ax
```

Note that the previous two forms (register or memory indirect) are really the same instruction. The mod and r/m fields of a mod-reg-r/m byte specify a register or memory indirect address. See Appendix D for the details.

The sixth form of the jmp instruction, the indirect intersegment jump, has a memory operand that contains a double word pointer. The CPU copies the double word at that address into the cs:ip register pair. For example,

```
FarPointer      dword   TargetAddress
                 .
                 .
                 .
                jmp     FarPointer
```

transfers control to the segmented address specified by the four bytes at address FarPointer. This instruction is semantically identical to the (mythical) instruction

```
                lcs ip, FarPointer    ;load cs, ip from FarPointer
```

As for the near indirect jump described earlier, this *far indirect* jump lets you specify any arbitrary (valid) memory addressing mode. You are not limited to the displacement only addressing mode the example above uses.

MASM uses a near indirect or far indirect addressing mode depending upon the type of the memory location you specify. If the variable you specify is a word variable, MASM will automatically generate a near indirect jump; if the variable is a dword, MASM emits the opcode for a far indirect jump. Some forms of memory addressing, unfortunately, do not intrinsically specify a size. For example, [bx] is definitely a memory operand, but does bx point at a word variable or a double word variable? It *could* point at either. Therefore, MASM will reject a statement of the form:

```
                jmp     [bx]
```

MASM cannot tell whether this should be a near indirect or far indirect jump. To resolve the ambiguity, you will need to use a *type coercion operator*. Chapter Eight will fully

16. Technically, this is syntactically incorrect because MASM cannot figure out the size of the memory operand. Read on for the details.

describe type coercion operators, for now, just use one of the following two instructions for a near or far jump, respectively:

```
jmp     word ptr [bx]
jmp     dword ptr [bx]
```

The register indirect addressing modes are not the only ones that could be type ambiguous. You could also run into this problem with indexed and base plus index addressing modes:

```
jmp     word ptr 5[bx]
jmp     dword ptr 9[bx][si]
```

For more information on the type coercion operators, see Chapter Eight.

In theory, you could use the indirect jump instructions and the set*cc* instructions to *conditionally* transfer control to some given location. For example, the following code transfers control to iftrue if word variable X is equal to word variable Y. It transfers control to iffalse, otherwise.

```
JmpTbl          word    iffalse, iftrue
                 .
                 .
                 .
                mov     ax, X
                cmp     ax, Y
                sete    bl
                movzx   ebx, bl
                jmp     JmpTbl[ebx*2]
```

As you will soon see, there is a *much* better way to do this using the conditional jump instructions.

6.9.2 The CALL and RET Instructions

The call and ret instructions handle subroutine calls and returns. There are five different call instructions and six different forms of the return instruction:

```
call    disp16      ;direct intrasegment, 16 bit relative.
call    adrs32      ;direct intersegment, 32 bit segmented address.
call    mem16       ;indirect intrasegment, 16 bit memory pointer.
call    reg16       ;indirect intrasegment, 16 bit register pointer.
call    mem32       ;indirect intersegment, 32 bit memory pointer.

ret                 ;near or far return
retn                ;near return
retf                ;far return
ret     disp        ;near or far return and pop
retn    disp        ;near return and pop
retf    disp        ;far return and pop
```

The call instructions take the same forms as the jmp instructions except there is no short (two byte) intrasegment call.

The far call instruction does the following:

- It pushes the cs register onto the stack.
- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 32 bit effective address into the cs:ip registers. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The near call instruction does the following:

- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 16 bit effective address into the ip register. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The call $disp_{16}$ instruction uses relative addressing. You can compute the effective address of the target by adding this 16 bit displacement with the return address (like the relative jmp instructions, the displacement is the distance from the instruction *following* the call to the target address).

The call $adrs_{32}$ instruction uses the direct addressing mode. A 32 bit segmented address immediately follows the call opcode. This form of the call instruction copies that value directly into the cs:ip register pair. In many respects, this is equivalent to the immediate addressing mode since the value this instruction copies into the cs:ip register pair immediately follows the instruction.

Call mem_{16} uses the memory indirect addressing mode. Like the jmp instruction, this form of the call instruction fetches the word at the specified memory location and uses that word's value as the target address. Remember, you can use *any* memory addressing mode with this instruction. The displacement-only addressing mode is the most common form, but the others are just as valid:

```
call      CallTbl[bx]           ;Index into an array of pointers.
call      word ptr [bx]         ;BX points at word to use.
call      WordTbl[bx][si]       ; etc.
```

Note that the selection of addressing mode only affects the effective address computation for the target subroutine. These call instructions still push the offset of the next instruction following the call onto the stack. Since these are *near* calls (they obtain their target address from a 16 bit memory location), they all push a 16 bit return address onto the stack.

Call reg_{16} works just like the memory indirect call above, except it uses the 16 bit value in a register for the target address. This instruction is really the same instruction as the call mem_{16} instruction. Both forms specify their effective address using a mod-reg-r/m byte. For the call reg_{16} form, the mod bits contain 11b so the r/m field specifies a register rather than a memory addressing mode. Of course, this instruction also pushes the 16 bit offset of the next instruction onto the stack as the return address.

The call mem_{32} instruction is a far indirect call. The memory address specified by this instruction must be a double word value. This form of the call instruction fetches the 32 bit segmented address at the computed effective address and copies this double word value into the cs:ip register pair. This instruction also copies the 32 bit segmented address of the next instruction onto the stack (it pushes the segment value first and the offset portion second). Like the call mem_{16} instruction, you can use any valid memory addressing mode with this instruction:

```
call      DWordVar
call      DwordTbl[bx]
call      dword ptr [bx]
etc.
```

It is relatively easy to synthesize the call instruction using two or three other 80x86 instructions. You could create the equivalent of a near call using a push and a jmp instruction:

```
push      <offset of instruction after jmp>
jmp       subroutine
```

A far call would be similar, you'd need to add a push cs instruction before the two instructions above to push a far return address on the stack.

The ret (return) instruction returns control to the caller of a subroutine. It does so by popping the return address off the stack and transferring control to the instruction at this

return address. Intrasegment (near) returns pop a 16 bit return address off the stack into the ip register. An intersegment (far) return pops a 16 bit offset into the ip register and then a 16 bit segment value into the cs register. These instructions are effectively equal to the following:

```
retn:           pop     ip
retf:           popd    cs:ip
```

Clearly, you must match a near subroutine call with a near return and a far subroutine call with a corresponding far return. If you mix near calls with far returns or vice versa, you will leave the stack in an inconsistent state and you probably will *not* return to the proper instruction after the call. Of course, another important issue when using the call and ret instructions is that you must make sure your subroutine doesn't push something onto the stack and then fail to pop it off before trying to return to the caller. Stack problems are a major cause of errors in assembly language subroutines. Consider the following code:

```
Subroutine:     push    ax
                push    bx
                  .
                  .
                  .
                pop     bx
                ret
                  .
                  .
                  .
                call    Subroutine
```

The call instruction pushes the return address onto the stack and then transfers control to the first instruction of subroutine. The first two push instructions push the ax and bx registers onto the stack, presumably in order to preserve their value because subroutine modifies them. Unfortunately, a programming error exists in the code above, subroutine only pops bx from the stack, it fails to pop ax as well. This means that when subroutine tries to return to the caller, the value of ax rather than the return address is sitting on the top of the stack. Therefore, this subroutine returns control to the address specified by the initial value of the ax register rather than to the true return address. Since there are 65,536 different values ax can have, there is a $^{1}/_{65,536}{}^{th}$ of a chance that your code will return to the real return address. The odds are not in your favor! Most likely, code like this will hang up the machine. Moral of the story – always make sure the return address is sitting on the stack before executing the return instruction.

Like the call instruction, it is very easy to simulate the ret instruction using two 80x86 instructions. All you need to do is pop the return address off the stack and then copy it into the ip register. For near returns, this is a very simple operation, just pop the near return address off the stack and then jump indirectly through that register:

```
                pop     ax
                jmp     ax
```

Simulating a far return is a little more difficult because you must load cs:ip in a single operation. The only instruction that does this (other than a far return) is the jmp mem_{32} instruction. See the exercises at the end of this chapter for more details.

There are two other forms of the ret instruction. They are identical to those above except a 16 bit displacement follows their opcodes. The CPU adds this value to the stack pointer immediately after popping the return address from the stack. This mechanism removes parameters pushed onto the stack before returning to the caller. See "Passing Parameters on the Stack" on page 581 for more details.

The assembler allows you to type ret without the "f" or "n" suffix. If you do so, the assembler will figure out whether it should generate a near return or a far return. See the chapter on procedures and functions for details on this.

6.9.3 The INT, INTO, BOUND, and IRET Instructions

The int (for software interrupt) instruction is a very special form of a call instruction. Whereas the call instruction calls subroutines within your program, the int instruction calls system routines and other special subroutines. The major difference between *interrupt service routines* and standard procedures is that you can have any number of different procedures in an assembly language program, while the *system* supports a maximum of 256 different interrupt service routines. A program calls a subroutine by specifying the *address* of that subroutine; it calls an interrupt service routine by specifying the *interrupt number* for that particular interrupt service routine. This chapter will only describe how to *call* an interrupt service routine using the int, into, and bound instructions, and how to return from an interrupt service routine using the iret instruction.

There are four different forms of the int instruction. The first form is

```
int       nn
```

(where "nn" is a value between 0 and 255). It allows you to call one of 256 different interrupt routines. This form of the int instruction is two bytes long. The first byte is the int opcode. The second byte is immediate data containing the interrupt number.

Although you can use the int instruction to call procedures (interrupt service routines) you've written, the primary purpose of this instruction is to make a *system call*. A system call is a subroutine call to a procedure provided by the system, such as a DOS , PC-BIOS[17], mouse, or some other piece of software resident in the machine before your program began execution. Since you always refer to a specific system call by its interrupt number, rather than its address, your program does not need to know the actual address of the subroutine in memory. The int instruction provides *dynamic linking* to your program. The CPU determines the actual address of an interrupt service routine at run time by looking up the address in an *interrupt vector table*. This allows the authors of such system routines to change their code (including the entry point) without fear of breaking any older programs that call their interrupt service routines. As long as the system call uses the same interrupt number, the CPU will automatically call the interrupt service routine at its new address.

The only problem with the int instruction is that it supports only 256 different interrupt service routines. MS-DOS alone supports well over 100 different calls. BIOS and other system utilities provide thousands more. This is above and beyond all the interrupts reserved by Intel for hardware interrupts and traps. The common solution most of the system calls use is to employ a *single* interrupt number for a given class of calls and then pass a *function number* in one of the 80x86 registers (typically the ah register). For example, MS-DOS uses only a single interrupt number, 21h. To choose a particular DOS function, you load a DOS *function code* into the ah register before executing the int 21h instruction. For example, to terminate a program and return control to MS-DOS, you would normally load ah with 4Ch and call DOS with the int 21h instruction:

```
mov       ah, 4ch       ;DOS terminate opcode.
int       21h           ;DOS call
```

The BIOS keyboard interrupt is another good example. Interrupt 16h is responsible for testing the keyboard and reading data from the keyboard. This BIOS routine provides several calls to read a character and scan code from the keyboard, see if any keys are available in the system type ahead buffer, check the status of the keyboard modifier flags, and so on. To choose a particular operation, you load the function number into the ah register before executing int 16h. The following table lists the possible functions:

17. BIOS stands for Basic Input/Output System.

Table 31: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	al = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay bl= 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the bl register contains the autorepeat rate.
5	ch = scan code cl = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in cl. This routine will simply insert the data you provide into the system type ahead buffer.
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).

Table 31: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows: bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10:Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

For example, to read a character from the system type ahead buffer, leaving the ASCII code in al, you could use the following code:

```
mov       ah, 0                ;Wait for key available, and then
int       16h                  ; read that key.
mov       character, al        ;Save character read.
```

Likewise, if you wanted to test the type ahead buffer to see if a key is available, *without reading that keystroke*, you could use the following code:

```
mov       ah, 1                ;Test to see if key is available.
int       16h                  ;Sets the zero flag if a key is not
                               ; available.
```

For more information about the PC-BIOS and MS-DOS, see “MS-DOS, PC-BIOS, and File I/O” on page 699.

The second form of the int instruction is a special case:

```
int       3
```

Int 3 is a special form of the interrupt instruction that is only one byte long. CodeView and other debuggers use it as a software breakpoint instruction. Whenever you set a breakpoint on an instruction in your program, the debugger will typically replace the first byte of the instruction’s opcode with an int 3 instruction. When your program executes the int 3 instruction, this makes a “system call” to the debugger so the debugger can regain control of the CPU. When this happens, the debugger will replace the int 3 instruction with the original opcode.

While operating inside a debugger, you can explicitly use the int 3 instruction to stop program executing and return control to the debugger. *This is not, however, the normal way to terminate a program*. If you attempt to execute an int 3 instruction while running under DOS, rather than under the control of a debugger program, you will likely crash the system.

The third form of the int instruction is into. Into will cause a software breakpoint if the 80x86 overflow flag is set. You can use this instruction to quickly test for arithmetic overflow after executing an arithmetic instruction. Semantically, this instruction is equivalent to

```
if overflow = 1 then int 4
```

You should *not* use this instruction unless you've supplied a corresponding trap handler (interrupt service routine). Doing so would probably crash the system. .

The fourth software interrupt, provided by 80286 and later processors, is the bound instruction. This instruction takes the form

```
bound     reg, mem
```

and executes the following algorithm:

```
if (reg < [mem]) or (reg > [mem+sizeof(reg)]) then int 5
```

[mem] denotes the contents of the memory location mem and sizeof(reg) is two or four depending on whether the register is 16 or 32 bits wide. The memory operand must be twice the size of the register operand. The bound instruction compares the values using a *signed* integer comparison.

Intel's designers added the bound instruction to allow a quick check of the range of a value in a register. This is useful in Pascal, for example, which checking array bounds validity and when checking to see if a subrange integer is within an allowable range. There are two problems with this instruction, however. On 80486 and Pentium/586 processors, the bound instruction is generally slower than the sequence of instructions it would replace[18]:

```
cmp       reg, LowerBound
jl        OutOfBounds
cmp       reg, UpperBound
jg        OutOfBounds
```

On the 80486 and Pentium/586 chips, the sequence above only requires four clock cycles assuming you can use the immediate addressing mode and the branches are not taken[19]; the bound instruction requires 7-8 clock cycles under similar circumstances and also assuming the memory operands are in the cache.

A second problem with the bound instruction is that it executes an int 5 if the specified register is out of range. IBM, in their infinite wisdom, decided to use the int 5 interrupt handler routine to print the screen. Therefore, if you execute a bound instruction and the value is out of range, the system will, by default, print a copy of the screen to the printer. If you replace the default int 5 handler with one of your own, pressing the PrtSc key will transfer control to your bound instruction handler. Although there are ways around this problem, most people don't bother since the bound instruction is so slow.

Whatever int instruction you execute, the following sequence of events follows:

- The 80x86 pushes the flags register onto the stack;
- The 80x86 pushes cs and then ip onto the stack;
- The 80x86 uses the interrupt number (into is interrupt #4, bound is interrupt #5) times four as an index into the interrupt vector table and copies the double word at that point in the table into cs:ip.

The int instructions vary from a call in two major ways. First, call instructions vary in length from two to six bytes long, whereas int instructions are generally two bytes long (int 3, into, and bound are the exceptions). Second, and most important, the int instruction pushes the flags and the return address onto the stack while the call instruction pushes only the return address. Note also that the int instructions always push a far return address (i.e., a cs value and an offset within the code segment), only the far call pushes this double word return address.

Since int pushes the flags onto the stack you must use a special return instruction, iret (interrupt return), to return from a routine called via the int instructions. If you return from an interrupt procedure using the ret instruction, the flags will be left on the stack upon returning to the caller. The iret instruction is equivalent to the two instruction sequence: ret, popf (assuming, of course, that you execute popf before returning control to the address pointed at by the double word on the top of the stack).

18. The next section describes the jg and jl instructions.

19. In general, one would hope that having a bounds violation is very rare.

The int instructions clear the trace (T) flag in the flags register. They do not affect any other flags. The iret instruction, by its very nature, can affect all the flags since it pops the flags from the stack.

6.9.4 The Conditional Jump Instructions

Although the jmp, call, and ret instructions provide transfer of control, they do not allow you to make any serious decisions. The 80x86's conditional jump instructions handle this task. The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if..then statement.

The conditional jumps test one or more flags in the flags register to see if they match some particular pattern (just like the set*cc* instructions). If the pattern matches, control transfers to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow, and zero flags. For example, after the execution of a shift left instruction, you could test the carry flag to determine if it shifted a one out of the H.O. bit of its operand. Likewise, you could test the condition of the zero flag after a test instruction to see if any specified bits were one. Most of the time, however, you will probably execute a conditional jump after a cmp instruction. The cmp instruction sets the flags so that you can test for less than, greater than, equality, etc.

Note: Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions. The following tables list all the aliases for a particular instruction. These tables also list out the opposite branches. You'll soon see the purpose of the opposite branches.

Table 32: Jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Table 33: Jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposite
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Table 34: Jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

On the 80286 and earlier, these instructions are all two bytes long. The first byte is a one byte opcode followed by a one byte displacement. Although this leads to very compact instructions, a single byte displacement only allows a range of ±128 bytes. There is a simple trick you can use to overcome this limitation on these earlier processors:

- Whatever jump you're using, switch to its opposite form. (given in the tables above).
- Once you've selected the opposite branch, use it to jump over a `jmp` instruction whose target address is the original target address.

For example, to convert:

```
jc      Target
```

to the long form, use the following sequence of instructions:

```
                jnc     SkipJmp
                jmp     Target
SkipJmp:
```

If the carry flag is clear (NC=no carry), then control transfers to label SkipJmp, at the same point you'd be if you were using the jc instruction above. If the carry flag is set when encountering this sequence, control will fall through the jnc instruction to the jmp instruction that will transfer control to Target. Since the jmp instruction allows 16 bit displacement and far operands, you can jump anywhere in the memory using this trick.

One brief comment about the "opposites" column is in order. As mentioned above, when you need to manually extend a branch from ±128 you should choose the opposite branch to branch around a jump to the target location. As you can see in the "aliases" column above, many conditional jump instructions have aliases. This means that there will be aliases for the opposite jumps as well. *Do not use any aliases when extending branches that are out of range*. With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the j*cc* instruction is *not* an "n", insert an "n" after the "j". E.g., je becomes jne and jl becomes jnl.
- If the second letter of the j*cc* instruction *is* an "n", then remove that "n" from the instruction. E.g., jng becomes jg, jne becomes je.

The two exceptions to this rule are jpe (jump parity even) and jpo (jump parity odd). These exceptions cause few problems because (a) you'll hardly ever need to test the parity flag, and (b) you can use the aliases jp and jnp synonyms for jpe and jpo. The "N/No N" rule applies to jp and jnp.

Though you *know* that jge is the opposite of jl, get in the habit of using jnl rather than jge. It's too easy in an important situation to start thinking "greater is the opposite of less" and substitute jg instead. You can avoid this confusion by always using the "N/No N" rule.

MASM 6.x and many other modern 80x86 assemblers will automatically convert out of range branches to this sequence for you. There is an option that will allow you to disable this feature. For performance critical code that runs on 80286 and earlier processors, you may want to disable this feature so you can fix the branches yourself. The reason is quite simple, this simple fix always wipes out the pipeline no matter which condition is true since the CPU jumps in either case. One thing nice about conditional jumps is that you do not flush the pipeline or the prefetch queue if you do not take the branch. If one condition is true far more often than the other, you might want to use the conditional jump to transfer control to a jmp nearby, so you can continue to fall through as before. For example, if you have a je target instruction and target is out of range, you could convert it to the following code:

```
                je      GotoTarget
                 .
                 .
                 .
GotoTarget:     jmp     Target
```

Although a branch to target now requires executing *two* jumps, this is much more efficient than the standard conversion if the zero flag is normally clear when executing the je instruction.

The 80386 and later processor provide an extended form of the conditional jump that is four bytes long, with the last two bytes containing a 16 bit displacement. These conditional jumps can transfer control anywhere within the current code segment. Therefore, there is no need to worry about manually extending the range of the jump. If you've told MASM you're using an 80386 or later processor, it will automatically choose the two byte or four byte form, as necessary. See Chapter Eight to learn how to tell MASM you're using an 80386 or later processor.

The 80x86 conditional jump instruction give you the ability to split program flow into one of two paths depending upon some logical condition. Suppose you want to increment the ax register if bx is or equal to cx. You can accomplish this with the following code:

```
                cmp     bx, cx
                jne     SkipStmts
                inc     ax
SkipStmts:
```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the "opposite branch (N/no N)" rule given earlier to select the opposite branch. You can make the same mistake choosing an opposite branch here as you could when extending out of range jumps.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the Enter key (carriage return):

```
                mov     di, 0
ReadLnLoop:     mov     ah, 0       ;INT 16h read key opcode.
                int     16h
                mov     Input[di], al
                inc     di
                cmp     al, 0dh     ;Carriage return ASCII code.
                jne     ReadLnLoop
                mov     Input[di-1],0;Replace carriage return with zero.
```

For more information concerning the use of the conditional jumps to synthesize IF statements, loops, and other control structures, see "Control Structures" on page 521.

Like the set*cc* instructions, the conditional jump instructions come in two basic categories – those that test specific process flag values (e.g., jz, jc, jno) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a cmp instruction. The cmp instruction sets the flags so you can use a ja, jae, jb, jbe, je, or jne instruction to test for unsigned less than, less than or equal, equality, inequality, greater than, or greater than or equal. Simultaneously, the cmp instruction sets the flags so you can also do a signed comparison using the jl, jle, je, jne, jg, and jge instructions.

The conditional jump instructions only test flags, they do not affect any of the 80x86 flags.

6.9.5 The JCXZ/JECXZ Instructions

The jcxz (jump if cx is zero) instruction branches to the target address if cx contains zero. Although you can use it anytime you need to see if cx contains zero, you would normally use it before a loop you've constructed with the loop instructions. The loop instruction can repeat a sequence of operations cx times. If cx equals zero, loop will repeat the operation 65,536 times. You can use jcxz to skip over such a loop when cx is zero.

The jecxz instruction, available only on 80386 and later processors, does essentially the same job as jcxz except it tests the full ecx register. Note that the jcxz instruction only checks cx, even on an 80386 in 32 bit mode.

There are no "opposite" jcxz or jecxz instructions. Therefore, you cannot use "N/No N" rule to extend the jcxz and jecxz instructions. The easiest way to solve this problem is to break the instruction up into two instructions that accomplish the same task:

```
                jcxz    Target
```

becomes

```
                test    cx, cx      ;Sets the zero flag if cx=0
                je      Target
```

Now you can easily extend the je instruction using the techniques from the previous section.

The test instruction above will set the zero flag if and only if cx contains zero. After all, if there are any non-zero bits in cx, logically anding them with themselves will produce a non-zero result. This is an efficient way to see if a 16 or 32 bit register contains zero. In fact, this two instruction sequence is *faster* than the jcxz instruction on the 80486 and later processors. Indeed, Intel recommends the use of this sequence rather than the jcxz instruction if you are concerned with speed. Of course, the jcxz instruction is shorter than the two instruction sequence, but it is not faster. This is a good example of an exception to the rule "shorter is usually faster."

The jcxz instruction does not affect any flags.

6.9.6 The LOOP Instruction

This instruction decrements the cx register and then branches to the target location if the cx register does not contain zero. Since this instruction decrements cx then checks for zero, if cx originally contained zero, any loop you create using the loop instruction will repeat 65,536 times. If you do not want to execute the loop when cx contains zero, use jcxz to skip over the loop.

There is no "opposite" form of the loop instruction, and like the jcxz/jecxz instructions the range is limited to ±128 bytes on all processors. If you want to extend the range of this instruction, you will need to break it down into discrete components:

```
; "loop lbl" becomes:

                dec     cx
                jne     lbl
```

You can easily extend this jne to any distance.

There is no eloop instruction that decrements ecx and branches if not zero (there is a loope instruction, but it does something else entirely). The reason is quite simple. As of the 80386, Intel's designers stopped wholeheartedly supporting the loop instruction. Oh, it's there to ensure compatibility with older code, but it turns out that the dec/jne instructions are actually *faster* on the 32 bit processors. Problems in the decoding of the instruction and the operation of the pipeline are responsible for this strange turn of events.

Although the loop instruction's name suggests that you would normally create loops with it, keep in mind that all it is really doing is decrementing cx and branching to the target address if cx does not contain zero after the decrement. You can use this instruction anywhere you want to decrement cx and then check for a zero result, not just when creating loops. Nonetheless, it is a very convenient instruction to use if you simply want to repeat a sequence of instructions some number of times. For example, the following loop initializes a 256 element array of bytes to the values 1, 2, 3, ...

```
                mov     ecx, 255
ArrayLp:        mov     Array[ecx], cl
                loop    ArrayLp
                mov     Array[0], 0
```

The last instruction is necessary because the loop does not repeat when cx is zero. Therefore, the last element of the array that this loop processes is Array[1], hence the last instruction.

The loop instruction does not affect any flags.

6.9.7 The LOOPE/LOOPZ Instruction

Loope/loopz (loop while equal / zero, they are synonyms for one another) will branch to the target address if cx is not zero and the zero flag is set. This instruction is quite useful

after cmp or cmps instruction, and is marginally faster than the comparable 80386/486 instructions *if you use all the features of this instruction*. However, this instruction plays havoc with the pipeline and superscalar operation of the Pentium so you're probably better off sticking with discrete instructions rather than using this instruction. This instruction does the following:

```
cx := cx - 1
if ZeroFlag = 1 and cx ≠ 0, goto target
```

The loope instruction falls through on one of two conditions. Either the zero flag is clear or the instruction decremented cx to zero. By testing the zero flag after the loop instruction (with a je or jne instruction, for example), you can determine the cause of termination.

This instruction is useful if you need to repeat a loop while some value is equal to another, but there is a maximum number of iterations you want to allow. For example, the following loop scans through an array looking for the first non-zero byte, but it does not scan beyond the end of the array:

```
                mov     cx, 16          ;Max 16 array elements.
                mov     bx, -1          ;Index into the array (note next inc).
SearchLp:       inc     bx              ;Move on to next array element.
                cmp     Array[bx], 0    ;See if this element is zero.
                loope   SearchLp        ;Repeat if it is.
                je      AllZero         ;Jump if all elements were zero.
```

Note that this instruction is not the opposite of loopnz/loopne. If you need to extend this jump beyond ±128 bytes, you will need to synthesize this instruction using discrete instructions. For example, if loope target is out of range, you would need to use an instruction sequence like the following:

```
                jne     quit
                dec     cx
                je      Quit2
                jmp     Target
quit:           dec     cx              ;loope decrements cx, even if ZF=0.
quit2:
```

The loope/loopz instruction does not affect any flags.

6.9.8 The LOOPNE/LOOPNZ Instruction

This instruction is just like the loope/loopz instruction in the previous section except loopne/loopnz (loop while not equal/not zero) repeats while cx is not zero and the zero flag is clear. The algorithm is

```
cx := cx - 1
if ZeroFlag = 0 and cx ≠ 0, goto target
```

You can determine if the loopne instruction terminated because cx was zero or if the zero flag was set by testing the zero flag immediately after the loopne instruction. If the zero flag is clear at that point, the loopne instruction fell through because it decremented cx to zero. Otherwise it fell through because the zero flag was set.

This instruction is *not* the opposite of loope/loopz. If the target address is out of range, you will need to use an instruction sequence like the following:

```
                je      quit
                dec     cx
                je      Quit2
                jmp     Target
quit:           dec     cx              ;loopne decrements cx, even if ZF=1.
quit2:
```

You can use the loopne instruction to repeat some maximum number of times while waiting for some other condition to be true. For example, you could scan through an array until you exhaust the number of array elements or until you find a certain byte using a loop like the following:

```
                mov     cx, 16      ;Maximum # of array elements.
                mov     bx, -1      ;Index into array.
LoopWhlNot0:    inc     bx          ;Move on to next array element.
                cmp     Array[bx],0 ;Does this element contain zero?
                loopne  LoopWhlNot0 ;Quit if it does, or more than 16 bytes.
```

Although the loope/loopz and loopne/loopnz instructions are slower than the individual instruction from which they could be synthesized, there is one main use for these instruction forms where speed is rarely important; indeed, being faster would make them less useful – timeout loops during I/O operations. Suppose bit #7 of input port 379h contains a one if the device is busy and contains a zero if the device is not busy. If you want to output data to the port, you *could* use code like the following:

```
                mov     dx, 379h
WaitNotBusy:    in      al, dx      ;Get port
                test    al, 80h     ;See if bit #7 is one
                jne     WaitNotBusy ;Wait for "not busy"
```

The only problem with this loop is that it is conceivable that it would loop forever. In a real system, a cable could come unplugged, someone could shut off the peripheral device, and any number of other things could go wrong that would hang up the system. Robust programs usually apply a *timeout* to a loop like this. If the device fails to become busy within some specified amount of time, then the loop exits and raises an error condition. The following code will accomplish this:

```
                mov     dx, 379h    ;Input port address
                mov     cx, 0       ;Loop 65,536 times and then quit.
WaitNotBusy:    in      al, dx      ;Get data at port.
                test    al, 80h     ;See if busy
                loopne  WaitNotBusy ;Repeat if busy and no time out.
                jne     TimedOut    ;Branch if CX=0 because we timed out.
```

You could use the loope/loopz instruction if the bit were zero rather than one.

The loopne/loopnz instruction does not affect any flags.

6.10 Miscellaneous Instructions

There are various miscellaneous instructions on the 80x86 that don't fall into any category above. Generally these are instructions that manipulate individual flags, provide special processor services, or handle privileged mode operations.

There are several instructions that directly manipulate flags in the 80x86 flags register. They are

- clc Clears the carry flag
- stc Sets the carry flag
- cmc Complements the carry flag
- cld Clears the direction flag
- std Sets the direction flag
- cli Clears the interrupt enable/disable flag
- sti Sets the interrupt enable/disable flag

Note: you should be careful when using the cli instruction in your programs. Improper use could lock up your machine until you cycle the power.

The nop instruction doesn't do anything except waste a few processor cycles and take up a byte of memory. Programmers often use it as a place holder or a debugging aid. As it turns out, this isn't a unique instruction, it's just a synonym for the xchg ax, ax instruction.

The hlt instruction halts the processor until a reset, non-maskable interrupt, or other interrupt (assuming interrupts are enabled) comes along. Generally, you shouldn't use this instruction on the IBM PC unless you really know what you are doing. *This instruction is not equivalent to the x86 halt instruction. Do not use it to stop your programs.*

The 80x86 provides another prefix instruction, lock, that, like the rep instruction, affects the following instruction. However, this instruction has little meaning on most PC systems. Its purpose is to coordinate systems that have multiple CPUs. As systems become available with multiple processors, this prefix *may* finally become valuable[20]. You need not be too concerned about this here.

The Pentium provides two additional instructions of interest to real-mode DOS programmers. These instructions are cpuid and rdtsc. If you load eax with zero and execute the cpuid instruction, the Pentium (and later processors) will return the maximum value cpuid allows as a parameter in eax. For the Pentium, this value is one. If you load the eax register with one and execute the cpuid instruction, the Pentium will return CPU identification information in eax. Since this instruction is of little value until Intel produces several additional chips in the family, there is no need to consider it further, here.

The second Pentium instruction of interest is the rdtsc (read time stamp counter) instruction. The Pentium maintains a 64 bit counter that counts clock cycles starting at reset. The rdtsc instruction copies the current counter value into the edx:eax register pair. You can use this instruction to accurately time sequences of code.

Besides the instructions presented thus far, the 80286 and later processors provide a set of *protected mode instructions*. This text will not consider those protected most instructions that are useful only to those who are writing operating systems. You would not even use these instructions in your applications when running under a protected mode operating system like Windows, UNIX, or OS/2. These instructions are reserved for the individuals who write such operating systems and drivers for them.

6.11 Sample Programs

The following sample programs demonstrate the use of the various instructions appearing in this chapter.

6.11.1 Simple Arithmetic I

```
; Simple Arithmetic
; 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
```

20. There are multiprocessor systems that have multiple Pentium chips installed. However, these systems generally use both CPUs only while running Windows NT, OS/2, or some other operating system that support symmetrical multiprocessing.

```
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
```

6.11.2 Simple Arithmetic II

```
; Simple Arithmetic
; 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
```

6.11.3 Logical Operations

```
; 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
```

6.11.4 Shift and Rotate Operations

```
; 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
```

6.11.5 Bit Operations and SETcc Instructions

```
; 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 "le" is for signed values.

; Result := u1 <= u2

                mov     ax, u1
                cmp     ax, u2
                setbe   Result      ;Note that "be" 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 "l" for signed comparisons.

                mov     ax, u1
                cmp     ax, u2
                seta    al          ;Use "a" 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
```

6.11.6 String Operations

```
; 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
```

6.11.7 Conditional Jumps

```
; 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"
; 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 Eight 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
```

6.11.8 CALL and INT Instructions

```
; 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 the chapter on interrupts for
; details on the interrupt vector table).
; The following call tells the PC's BIOS
; to print theASCII 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

                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
```

6.11.9 Conditional Jumps I

```
; 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)
lbl2:
                jna     Temp
                jmp     lbl3
Temp:
                byte    150 dup (0)
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)
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
```

6.11.10 Conditional Jump Instructions II

```
; 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
```

6.12 Laboratory Exercises

In this set of laboratory exercises you will be writing programs in IBM/L – *Instruction BenchMarking Language.* The IBM/L system lets you time certain instruction sequences to see how long they take to execute.

6.12.1 The IBM/L System

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:

```
#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
```

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 Chapter Seven.

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
```

6.12.2 IBM/L Exercises

The Chapter Six directory contains several sample IBM/L programs (the *.ibm files).. Ex6_1.ibm tests three sequences that compute the absolute value of an integer. Ex6_2.ibm tests three different ways to do a shl left by eight bits. Ex6_3.ibm tests accessing word data at even and odd addresses. Ex6_4.ibm compares the amount of time it takes to load es:bx from a memory location with the time it takes to load es:bx with a constant. Ex6_5.ibm compares the amount of time it takes to swap two registers with and without the XCHG instruction. Ex6_6.ibm compares the multiply instruction against shift & add versions. Ex6_7.ibm compares the speed of register-register move instruction against register-memory move instructions.

Compile each of these IBM/L programs with a DOS command that takes the form:

```
ibml ex6_1.ibm >ex6_1.asm
```

For your lab report: IBM/L writes its output to the standard output device. So use the redirection operator to send this output to a file. Once you've created the file, assemble it with MASM and execute the result. Include the IBM/L program listing and the results in your lab report. **For additional credit:** write your own IBM/L programs to test certain instruction sequences. Include the IBM/L source code in your lab report along with your results.

Warning: to get the most accurate results, you should not run the assembly language programs IBM/L creates under Windows or any other multitasking operating system. For best results, run the output of IBM/L under DOS.

6.13 Programming Projects

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 80^{th} 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 companion CD-ROM (proj6_1.asm).

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).

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.

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 companion CD-ROM (proj6_4.asm).

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 companion CD-ROM to begin this project (proj6_5.asm).

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_6.asm to start this project.

6.14 Summary

The 80x86 processor family provides a rich CISC (complex instruction set computer) instruction set. Members of the 80x86 processor family are generally *upward compatible*, meaning successive processors execute all the instructions of the previous chips. Programs written for an 80x86 will generally run on all members of the family, programs using new instructions on the 80286 will run on the 80286 and later processors, but not on the 8086. Likewise, programs that take advantage of the new instructions on the 80386 will run on the 80386 and later processors, but not on the earlier processors. And so on.

The processors described in this chapter include the 8086/8088, the 80286, the 80386, the 80486, and the Pentium (80586). Intel also produced an 80186, but this processor was not used extensively in personal computers[21].

The 80x86 instruction set is most easily broken down into eight categories, see

- "Data Movement Instructions" on page 246.
- "Conversions" on page 252.
- "Arithmetic Instructions" on page 255.
- "Logical, Shift, Rotate and Bit Instructions" on page 269.
- "I/O Instructions" on page 284.
- "String Instructions" on page 284.

21. A few PCs actually used this CPU. Control applications were the biggest user of this CPU, however. The 80186 includes most of the 80286 specific instructions described in this chapter. It does not include the protected mode instructions of the 80286.

Many instructions affect various flag bits in the 80x86 flags register. Some instructions can test these flags as though they were boolean values. The flags also denote relationships after a comparison such as equality, less than, and greater than. To learn about these flags and how to test them in your programs, consult

There are several instructions on the 80x86 that transfer data between registers and memory. These instructions are the ones assembly language programmers use most often. The 80x86 provides many such instructions to help you write fast, efficient, programs. For the details, read

The 80x86 provides several instructions to convert data from one form to another. There are special instructions for sign extension, zero extension, table translation, and big/little endian conversion.

The 80x86 arithmetic instructions provide all the common integer operations: addition, multiplication, subtraction, division, negation, comparisons, increment, decrement, and various instructions to help with BCD arithmetic: AAA, AAD, AAM, AAS, DAA, and DAS. For information on these instructions, see

The 80x86 also provides a rich set of logical, shift, rotate, and bit operations. These instructions manipulate the bits in their operands allowing you to logically AND, OR, XOR, and NOT values, rotate and shift bits within an operand, test and set/clear/invert bits in an operand, and set an operand to zero or one depending on the state of the flags register. For more information, see

There are a couple of I/O instruction in the 80x86 instruction set, IN and OUT. These are really special forms of the MOV instruction that operate on the 80x86 I/O address space rather than the memory address space. You typically use these instructions to access hardware registers on peripheral devices. This chapter discusses these instruction at

- "I/O Instructions" on page 284.

The 80x86 family provides a large repertoire of instructions that manipulate strings of data. These instructions include movs, lods, stos, scas, cmps, ins, outs, rep, repz, repe, repnz, and repne. For more information, see

- "String Instructions" on page 284

The transfer of control instructions on the 80x86 let you create loops, subroutines, conditional sequences and do many other tests. To learn about these instructions, read

- "Program Flow Control Instructions" on page 286.
- "Unconditional Jumps" on page 286.
- "The CALL and RET Instructions" on page 289.
- "The INT, INTO, BOUND, and IRET Instructions" on page 292.
- "The Conditional Jump Instructions" on page 296.
- "The JCXZ/JECXZ Instructions" on page 299.
- "The LOOP Instruction" on page 300.
- "The LOOPE/LOOPZ Instruction" on page 300.
- "The LOOPNE/LOOPNZ Instruction" on page 301

This chapter finally discusses various miscellaneous instructions. These instructions directly manipulate flags in the flags register, provide certain processor services, or perform protected mode operations. This Chapter only mentioned the protected mode instructions. Since you do not normally use them in application (non-O/S) programs, you don't really need to know much about them. See

- "Miscellaneous Instructions" on page 302

6.15 Questions

1) Provide an example that shows that it requires n+1 bits to hold the sum of two n-bit binary values.

2) ADC and SBB can be forced to behave exactly like ADD and SUB by inserting some other instruction before ADC and SBB. What instruction must be inserted in front of ADC to make it behave like ADD? In front of SBB to make it behave like SUB?

3) Given that you can manipulate data items on the top of stack using PUSH and POP, explain how you could modify a return address on the top of stack so that a RET instruction will cause the 80x86 to return two bytes beyond the original return address.

4) Provide four different ways to add two to the value in the BX register. No way should require more than two instructions (hint, there are at least six ways of doing this!)

5) Assume that the target addresses for the following conditional jumps are beyond the range of a short branch. Modify each of these instructions so that they perform the proper operation (i.e., can jump the entire distance):

   a) JS Label1 b) JE Label2 c) JZ Label3

   d) JC Label4 e) JBE There f) JG Label5

6) Explain the difference between the carry flag and the overflow flag.

7) When do you typically use the CBW and CWD instructions to sign extend a value?

8) What is the difference between a "MOV reg, immediate" instruction and a "LEA reg, address" instruction?

9) What does the INT nn instruction push onto the stack that the CALL FAR instruction does not?

10) What is the JCXZ instruction typically used for?

11) Explain the operation of the LOOP, LOOPE/LOOPZ, and LOOPNE/LOOPNZ instructions.

12) Which registers (other than the flag register) are affected by the MUL, IMUL, DIV, and IDIV instructions?

13) List three differences between the "DEC AX" and the "SUB AX, 1" instructions.

14) Which of the shift, rotate, and logical instructions do not affect the zero flag?

15) Why does the SAR instruction always clear the overflow flag?

16) On the 80386 the IMUL instruction is almost totally orthogonal (generalized). Almost. Give some examples of forms allowed by the ADD instruction for which there are no comparable IMUL instructions.

17) Why didn't Intel generalize the IDIV instruction as they did the IMUL instruction?

18) What instruction(s) would you need to use to read the eight bit value at I/O address 378h? Please give the specific instructions to do this.

19) Which flag(s) does the 80x86 use to check for *unsigned* arithmetic overflow?

20) Which flag(s) let you check for *signed* overflow?.

21) Which flag(s) does the 80x86 use to test the following *unsigned* conditions? How must the flags be set for the condition to be true?

   a) equal b) not equal c) less than

   d) less than or equal e) greater than f) greater than or equal

22) Repeat the above question for a *signed* comparison.

23) Explain the operation of the 80x86 CALL and RET instructions. Describe step-by-step what each variant of these instructions.

24) The following sequence exchanges the values between the two memory variables I and J:

```
xchg    ax, i
xchg    ax, j
xchg    ax, i
```

On the 80486, the "MOV reg, mem" and "MOV mem, reg" instructions take one cycle (under the right conditions) whereas the "XCHG reg, mem" instruction takes three cycles. Provide a faster sequence for the '486 than the above.

25) On the 80386 the "MOV reg, mem" instruction requires four cycles, the "MOV mem, reg" requires two cycles, and the "XCHG reg, mem" instruction requires five cycles. Provide a faster sequence of the memory exchange problem in question 24 for the 80386.

26) On the 80486, the "MOV acc, mem" and "MOV reg, mem" instructions all take only one cycle to execute (under the right conditions). Assuming all other things equal, why would you want to use the "MOV acc,mem" form rather than the "MOV reg,mem" form to load a value into AL/AX/EAX?

27) Which instructions perform 32 bit loads on the pre-80386 processors?

28) How could you use the PUSH and POP instructions to preserve the AX register between two points in your code?

29) If, immediately upon entering a subroutine, you execute a "pop ax" instruction, what value will you have in the AX register?

30) What is one major use for the SAHF instruction?

31) What is the difference between CWD and CWDE?

32) The BSWAP instruction will convert 32 bit *big endian* values to 32 bit *little endian* values. What instruction can you use to convert 16 bit big endian to 16 bit little endian values?

33) What instruction could you use to convert 32 bit little endian values to 32 bit big endian values?

34) Explain how you could use the XLAT instruction to convert an alphabetic character in the AL register from lower case to upper case (assuming it is lower case) and leave all other values in AL unchanged.

35) What instruction is CMP most similar to?

36) What instruction is TEST most similar to?

37) What does the NEG instruction do?

38) Under what two circumstances will the DIV and IDIV instructions fail?

39) What is the difference between RCL and ROL?

40) Write a short code segment, using the LOOP instruction, that calls the "CallMe" subroutine 25 times.

41) On the 80486 and Pentium CPUs the LOOP instruction is not as fast as the discrete instructions that perform the same operation. Rewrite the code above to produce a faster executing program on the 80486 and Pentium chips.

42) How do you determine the "opposite jump" for a conditional jump. Why is this algorithm preferable?

43) Give an example of the BOUND instruction. Explain what your example will do.

44) What is the difference between the IRET and RET (far) instructions?

45) The BT (Bit Test) instruction copies a specific bit into the carry flag. If the specified bit is one, it sets the carry flag, if the bit is zero, it clears the carry flag. Suppose you want to clear the carry flag if the bit was zero and set it otherwise. What instruction could you execute after BT to accomplish this?

46) You can simulate a far return instruction using a double word variable and two 80x86 instructions. What is the two instruction sequence that will accomplish this?