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, reg1
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 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
```

Reg16 is any general purpose 16 bit register and mem32 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 mem32 into reg16 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.