Lecture No.8

Lecture Outlines

4.1 Data Transfer Instructions

4.1.1 Introduction

When programming in languages like Java or C++, it's easy for beginners to be annoyed when the compilers generate lots of syntax error messages. Compilers perform strict type checking in order to help you avoid possible errors such as mismatching variables and data. Assemblers, on the other hand, let you do just about anything you want, as long as the processor's instruction set can do what you ask. In other words, assembly language forces you to pay attention to data storage and machine-specific details. You must understand the processor's limitations when you write assembly language code. As it happens, x86 processors have what is commonly known as a *complex instruction set,* so they offer a lot of ways of doing things.

4.1.2 Operand Types

Chapter 3 introduced x86 instruction formats:

```
[label:] mnemonic [operands][ ; comment ]
```

Instructions can have zero, one, two, or three operands. Here, we omit the label and comment fields for clarity:

```
mnemonic
mnemonic [destination]
mnemonic [destination],[source]
mnemonic [destination],[source-1],[source-2]
```

There are three basic types of operands:

- Immediate—uses a numeric literal expression
- Register—uses a named register in the CPU
- Memory—references a memory location

Table 4-1 describes the standard operand types. It uses a simple notation for operands (in 32-bit mode) freely adapted from the Intel manuals. We will use it from this point on to describe the syntax of individual instructions.

4.1.3 Direct Memory Operands

Variable names are references to offsets within the data segment. For example, the following declaration for a variable named **var1** says that its size attribute is **byte** and it contains the value 10 hexadecimal:

Table 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

```
.data
var1 BYTE 10h
```

We can write instructions that dereference (look up) memory operands using their addresses. Suppose **var1** were located at offset 10400h. The following instruction copies its value into the AL register:

```
mov al var1
```

It would be assembled into the following machine instruction:

```
A0 00010400
```

The first byte in the machine instruction is the operation code (known as the *opcode*). The remaining part is the 32-bit hexadecimal address of **var1**. Although it might be possible to write programs using only numeric addresses, symbolic names such as **var1** make it easier to reference memory.

Alternative Notation. Some programmers prefer to use the following notation with direct operands because the brackets imply a dereference operation:

```
mov  al,[var1]
```

MASM permits this notation, so you can use it in your own programs if you want. Because so many programs (including those from Microsoft) are printed without the brackets, we will only use them in this book when an arithmetic expression is involved:

```
mov  al,[var1 + 5]
```

(This is called a direct-offset operand, a subject discussed at length in Section 4.1.8.)

4.1.4 MOV Instruction

The MOV instruction copies data from a source operand to a destination operand. Known as a *data transfer* instruction, it is used in virtually every program. Its basic format shows that the first operand is the destination and the second operand is the source:

```
MOV destination,source
```

The destination operand's contents change, but the source operand is unchanged. The right to left movement of data is similar to the assignment statement in C++ or Java:

```
dest = source;
```

In nearly all assembly language instructions, the left-hand operand is the destination and the right-hand operand is the source. MOV is very flexible in its use of operands, as long as the following rules are observed:

- Both operands must be the same size.
- Both operands cannot be memory operands.
- The instruction pointer register (IP, EIP, or RIP) cannot be a destination operand.

Here is a list of the standard MOV instruction formats:

```
MOV reg,reg
MOV mem,reg
MOV reg,mem
MOV mem,imm
MOV reg,imm
```

Memory to Memory A single MOV instruction cannot be used to move data directly from one memory location to another. Instead, you must move the source operand's value to a register before assigning its value to a memory operand:

```
.data
var1 WORD ?
var2 WORD ?
.code
mov  ax,var1
mov  var2,ax
```

You must consider the minimum number of bytes required by an integer constant when copying it to a variable or register. For unsigned integer constant sizes, refer to Table 1-4 in Chapter 1. For signed integer constants, refer to Table 1-7.

Overlapping Values

The following code example shows how the same 32-bit register can be modified using differently sized data. When **oneWord** is moved to AX, it overwrites the existing value of AL. When **oneDword** is moved to EAX, it overwrites AX. Finally, when 0 is moved to AX, it overwrites the lower half of EAX.

```
.data
oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h
```

```
.code
mov  eax,0               ; EAX = 00000000h
mov  al,oneByte          ; EAX = 00000078h
mov  ax,oneWord          ; EAX = 00001234h
mov  eax,oneDword        ; EAX = 12345678h
mov  ax,0                ; EAX = 12340000h
```

4.1.5 Zero/Sign Extension of Integers

Copying Smaller Values to Larger Ones

Although MOV cannot directly copy data from a smaller operand to a larger one, programmers can create workarounds. Suppose **count** (unsigned, 16 bits) must be moved to ECX (32 bits). We can set ECX to zero and move **count** to CX:

```
.data
count WORD 1
.code
mov ecx,0
mov cx,count
```

What happens if we try the same approach with a signed integer equal to −16?

```
.data
signedVal SWORD -16              ; FFF0h (-16)
.code
mov ecx,0
mov cx,signedVal                 ; ECX = 0000FFF0h (+65,520)
```

The value in ECX (+65,520) is completely different from −16. On the other hand, if we had filled ECX first with FFFFFFFFh and then copied **signedVal** to CX, the final value would have been correct:

```
mov ecx,0FFFFFFFFh
mov cx,signedVal                 ; ECX = FFFFFFF0h (-16)
```

The effective result of this example was to use the highest bit of the source operand (1) to fill the upper 16 bits of the destination operand, ECX. This technique is called *sign extension*. Of course, we cannot always assume that the highest bit of the source is a 1. Fortunately, the engineers at Intel anticipated this problem when designing the instruction set and introduced the MOVZX and MOVSX instructions to deal with both unsigned and signed integers.

MOVZX Instruction

The MOVZX instruction (*move with zero-extend*) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants:

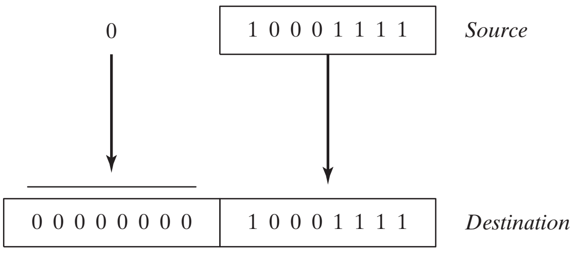
```
MOVZX  reg32,reg/mem8
MOVZX  reg32,reg/mem16
MOVZX  reg16,reg/mem8
```

(Operand notation was explained in Table 4-1.) In each of the three variants, the first operand (a register) is the destination and the second is the source. Notice that the source operand cannot be a constant. The following example zero-extends binary 10001111 into AX:

```
.data
byteVal BYTE 10001111b
.code
movzx  ax,byteVal                ; AX = 0000000010001111b
```

Figure 4-1 shows how the source operand is zero-extended into the 16-bit destination.

Figure 4–1 Using MOVZX to copy a byte into a 16-bit destination.



The following examples use registers for all operands, showing all the size variations:

```
mov    bx,0A69Bh
movzx  eax,bx                   ; EAX = 0000A69Bh
movzx  edx,bl                   ; EDX = 0000009Bh
movzx  cx,bl                    ; CX  = 009Bh
```

The following examples use memory operands for the source and produce the same results:

```
.data
byte1  BYTE 9Bh
word1  WORD 0A69Bh
.code
movzx  eax,word1                ; EAX = 0000A69Bh
movzx  edx,byte1                ; EDX = 0000009Bh
movzx  cx,byte1                 ; CX  = 009Bh
```

MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

```
MOVSX  reg32,reg/mem8
MOVSX  reg32,reg/mem16
MOVSX  reg16,reg/mem8
```

An operand is sign-extended by taking the smaller operand's highest bit and repeating (replicating) the bit throughout the extended bits in the destination operand. The following example sign-extends binary 10001111b into AX:

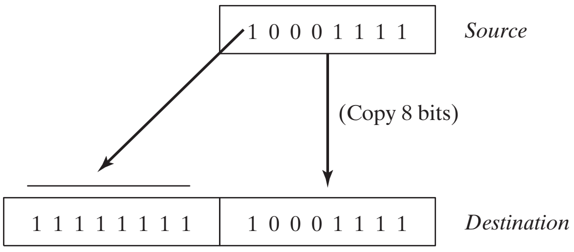
```
.data
byteVal BYTE 10001111b
.code
movsx  ax,byteVal                 ; AX = 1111111110001111b
```

The lowest 8 bits are copied as in Figure 4-2. The highest bit of the source is copied into each of the upper 8 bit positions of the destination.

A hexadecimal constant has its highest bit set if its most significant hexadecimal digit is greater than 7. In the following example, the hexadecimal value moved to BX is A69B, so the leading "A" digit tells us that the highest bit is set. (The leading zero appearing before A69B is just a notational convenience so the assembler does not mistake the constant for the name of an identifier.)

```
mov    bx,0A69Bh
movsx  eax,bx                     ; EAX = FFFFA69Bh
movsx  edx,bl                     ; EDX = FFFFFF9Bh
movsx  cx,bl                      ; CX  = FF9Bh
```

Figure 4–2 Using MOVSX to copy a byte into a 16-bit destination.



4.1.6 LAHF and SAHF Instructions

The LAHF (load status flags into AH) instruction copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry. Using this instruction, you can easily save a copy of the flags in a variable for safekeeping:

```
.data
saveflags BYTE ?
.code
lahf                              ; load flags into AH
mov  saveflags,ah                 ; save them in a variable
```

The SAHF (store AH into status flags) instruction copies AH into the low byte of the EFLAGS (or RFLAGS) register. For example, you can retrieve the values of flags saved earlier in a variable:

```
mov  ah,saveflags                 ; load saved flags into AH
sahf                              ; copy into Flags register
```

4.1.7 XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands. There are three variants:

```
XCHG  reg,reg
XCHG  reg,mem
XCHG  mem,reg
```

The rules for operands in the XCHG instruction are the same as those for the MOV instruction (Section 4.1.4), except that XCHG does not accept immediate operands. In array sorting applications, XCHG provides a simple way to exchange two array elements. Here are a few examples using XCHG:

```
xchg  ax,bx                ; exchange 16-bit regs
xchg  ah,al                ; exchange 8-bit regs
xchg  var1,bx              ; exchange 16-bit mem op with BX
xchg  eax,ebx              ; exchange 32-bit regs
```

To exchange two memory operands, use a register as a temporary container and combine MOV with XCHG:

```
mov   ax,val1
xchg  ax,val2
mov   val1,ax
```

4.1.8 Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels. Let's begin with an array of bytes named **arrayB**:

```
arrayB  BYTE 10h,20h,30h,40h,50h
```

If we use MOV with **arrayB** as the source operand, we automatically move the first byte in the array:

```
mov  al,arrayB                     ; AL = 10h
```

We can access the second byte in the array by adding 1 to the offset of **arrayB**:

```
mov  al,[arrayB+1]                 ; AL = 20h
```

The third byte is accessed by adding 2:

```
mov  al,[arrayB+2]                 ; AL = 30h
```

An expression such as **arrayB+1** produces what is called an *effective address* by adding a constant to the variable's offset. Surrounding an effective address with brackets makes it clear that the expression is dereferenced to obtain the contents of memory at the address. The assembler does not require you to surround address expressions with brackets, but we highly recommend their use for clarity.

MASM has no built-in range checking for effective addresses. In the following example, assuming **arrayB** holds five bytes, the instruction retrieves a byte of memory outside the array. The result is a sneaky logic bug, so be extra careful when checking array references:

```
mov  al,[arrayB+20]                ; AL = ??
```

Word and Doubleword Arrays In an array of 16-bit words, the offset of each array element is 2 bytes beyond the previous one. That is why we add 2 to **ArrayW** in the next example to reach the second element:

```
.data
arrayW WORD 100h,200h,300h
.code
mov  ax,arrayW                   ; AX = 100h
mov  ax,[arrayW+2]               ; AX = 200h
```

Similarly, the second element in a doubleword array is 4 bytes beyond the first one:

```
.data
arrayD DWORD 10000h,20000h
.code
mov  eax,arrayD                  ; EAX = 10000h
mov  eax,[arrayD+4]              ; EAX = 20000h
```

4.1.9 Example Program (Moves)

Let's combine all the instructions we've covered so far in this chapter, including MOV, XCHG, MOVSX, and MOVDX, to show how bytes, words, and doublewords are affected. We will also include some direct-offset operands.

```
; Data Transfer Examples            (Moves.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD
.data
val1 WORD 1000h
val2 WORD 2000h
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h
arrayD DWORD 10000h,20000h

.code
main PROC

;  Demonstrating MOVZX instruction:
    mov   bx,0A69Bh
    movzx eax,bx                      ; EAX = 0000A69Bh
    movzx edx,bl                      ; EDX = 0000009Bh
    movzx cx,bl                       ; CX  = 009Bh

;  Demonstrating MOVSX instruction:
    mov   bx,0A69Bh
    movsx eax,bx                      ; EAX = FFFFA69Bh
    movsx edx,bl                      ; EDX = FFFFFF9Bh
    mov   bl,7Bh
    movsx cx,bl                       ; CX  = 007Bh

;  Memory-to-memory exchange:
    mov   ax,val1                     ; AX = 1000h
```

```
        xchg  ax,val2                     ; AX=2000h, val2=1000h
        mov   val1,ax                     ; val1 = 2000h

    ;  Direct-Offset Addressing (byte array):
        mov   al,arrayB                   ; AL = 10h
        mov   al,[arrayB+1]               ; AL = 20h
        mov   al,[arrayB+2]               ; AL = 30h

    ;  Direct-Offset Addressing (word array):
        mov   ax,arrayW                   ; AX = 100h
        mov   ax,[arrayW+2]               ; AX = 200h

    ;  Direct-Offset Addressing (doubleword array):
         mov   eax,arrayD                 ; EAX = 10000h
         mov   eax,[arrayD+4]             ; EAX = 20000h
         mov   eax,[arrayD+4]             ; EAX = 20000h

         INVOKE ExitProcess,0
    main ENDP
    END main
```

This program generates no screen output, but you can (and should) run it using a debugger.

Displaying CPU Flags in the Visual Studio Debugger

To display the CPU status flags during a debugging session, select *Windows* from the *Debug* menu, then select *Registers* from the *Windows* menu. Inside the *Registers* window, right-click and select *Flags* from the dropdown list. You must be currently debugging a program in order to see these menu options. The following table identifies the flag symbols used inside the *Registers* window:

Flag Name	**Overflow**	**Direction**	**Interrupt**	**Sign**	**Zero**	**Aux Carry**	**Parity**	**Carry**
Symbol	OV	UP	EI	PL	ZR	AC	PE	CY

Each flag is assigned a value of 0 (*clear*) or 1 (*set*). Here's an example:

```
OV = 0  UP = 0  EI = 1
PL = 0  ZR = 1  AC = 0
PE = 1  CY = 0
```

As you step through your code during a debugging session, each flag displays in red when an instruction modifies the flag's value. You can learn how instructions affect the flags by stepping through instructions and keeping an eye on the changing values of the flags.

4.2 Addition and Subtraction

Arithmetic is a surprisingly big topic in assembly language! This chapter will focus on addition and subtraction. Then we will talk about multiplication and division later in Chapter 7. Then we'll switch over to floating point arithmetic in Chapter 12.

Let's start with the easiest and most efficient instructions of them all: INC (increment) and DEC (decrement), which add 1 and subtract 1. Then we will move on to the ADD, SUB, and NEG (negate) instructions, which offer more possibilities. Last of all, we will get into a discussion about how the CPU status flags (Carry, Sign, Zero, etc.) are affected by arithmetic instructions. Remember, assembly language is all about the details.

4.2.1 INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a register or memory operand. The syntax is

```
INC reg/mem
DEC reg/mem
```

Following are some examples:

```
.data
myWord WORD 1000h
.code
inc  myWord                    ; myWord = 1001h
mov  bx,myWord
dec  bx                        ; BX = 1000h
```

The Overflow, Sign, Zero, Auxiliary Carry, and Parity flags are changed according to the value of the destination operand. The INC and DEC instructions do not affect the Carry flag (which is something of a surprise).

4.2.2 ADD Instruction

The ADD instruction adds a source operand to a destination operand of the same size. The syntax is

```
ADD dest,source
```

Source is unchanged by the operation, and the sum is stored in the destination operand. The set of possible operands is the same as for the MOV instruction (Section 4.1.4). Here is a short code example that adds two 32-bit integers:

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov  eax,var1                  ; EAX = 10000h
add  eax,var2                  ; EAX = 30000h
```

Flags The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand. We will explain how the flags work in Section 4.2.6.

4.2.3 SUB Instruction

The SUB instruction subtracts a source operand from a destination operand. The set of possible operands is the same as for the ADD and MOV instructions. The syntax is

```
SUB dest,source
```

Here is a short code example that subtracts two 32-bit integers:

```
.data
var1 DWORD 30000h
var2 DWORD 10000h
.code
mov  eax,var1                  ; EAX = 30000h
sub  eax,var2                  ; EAX = 20000h
```

Flags The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

4.2.4 NEG Instruction

The NEG (negate) instruction reverses the sign of a number by converting the number to its two's complement. The following operands are permitted:

```
NEG reg
NEG mem
```

(Recall that the two's complement of a number can be found by reversing all the bits in the destination operand and adding 1.)

Flags The Carry, Zero, Sign, Overflow, Auxiliary Carry, and Parity flags are changed according to the value that is placed in the destination operand.

4.2.5 Implementing Arithmetic Expressions

Armed with the ADD, SUB, and NEG instructions, you have the means to implement arithmetic expressions involving addition, subtraction, and negation in assembly language. In other words, you can simulate what a C++ compiler might do when a statement such as this:

```
Rval = -Xval + (Yval - Zval);
```

Let's see how the sample statement would be implemented in assembly language. The following signed 32-bit variables will be used:

```
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 40
```

When translating an expression, evaluate each term separately and combine the terms at the end. First, we negate a copy of **Xval** and store it in a register:

```
; first term: -Xval
mov  eax,Xval
neg  eax                        ; EAX = -26
```

Then **Yval** is copied to a register and **Zval** is subtracted:

```
; second term: (Yval - Zval)
mov  ebx,Yval
sub  ebx,Zval                   ; EBX = -10
```

Finally, the two terms (in EAX and EBX) are added:

```
; add the terms and store:
add  eax,ebx
mov  Rval,eax                   ; -36
```