COAL Chapter 4

Week 3,4

Shoaib Rauf

Courtesy: Dr. Muhammad Nouman Durrani

Data Transfer Instructions

- **Operand Types**

x86 instruction formats:

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

Because the number of operands may vary, we can further subdivide the formats to 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*]

To give added flexibility to the instruction set, x86 assembly language uses different types of instruction operands. The following are the easiest to use:

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

MOV Instruction

- The MOV instruction copies data from a source operand to a destination operand.
- Its basic format shows that the first operand is the destination and the second operand is the source:

  MOV *destination, source*

Here is a list of the general variants of MOV, excluding segment registers:

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

Segment registers should not be directly modified by programs running in protected mode. The following options are available when running in real mode, with the exception that CS cannot be a target operand:

```
MOV reg/mem16,sreg
MOV sreg,reg/mem16
```

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 moving its value to a memory operand:

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

Overlapping Values:

```
.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
```

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)
```

Solution:

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

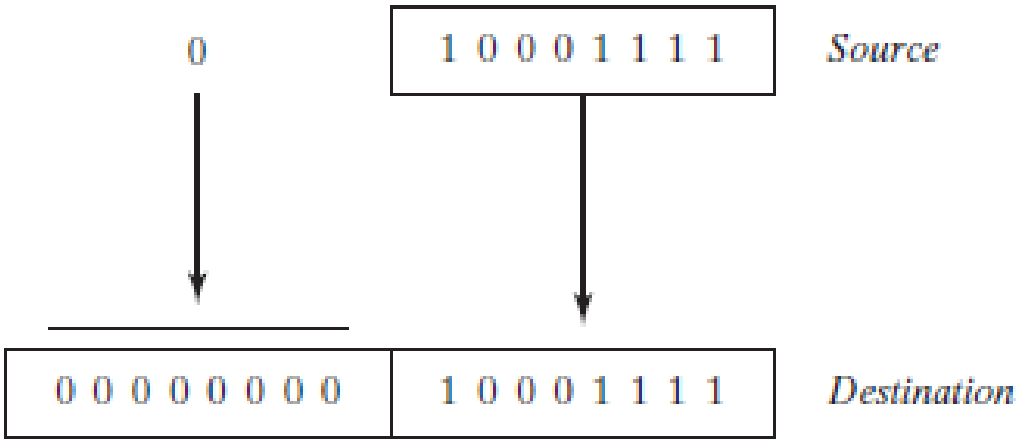
MOVZX Instruction

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
```

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

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

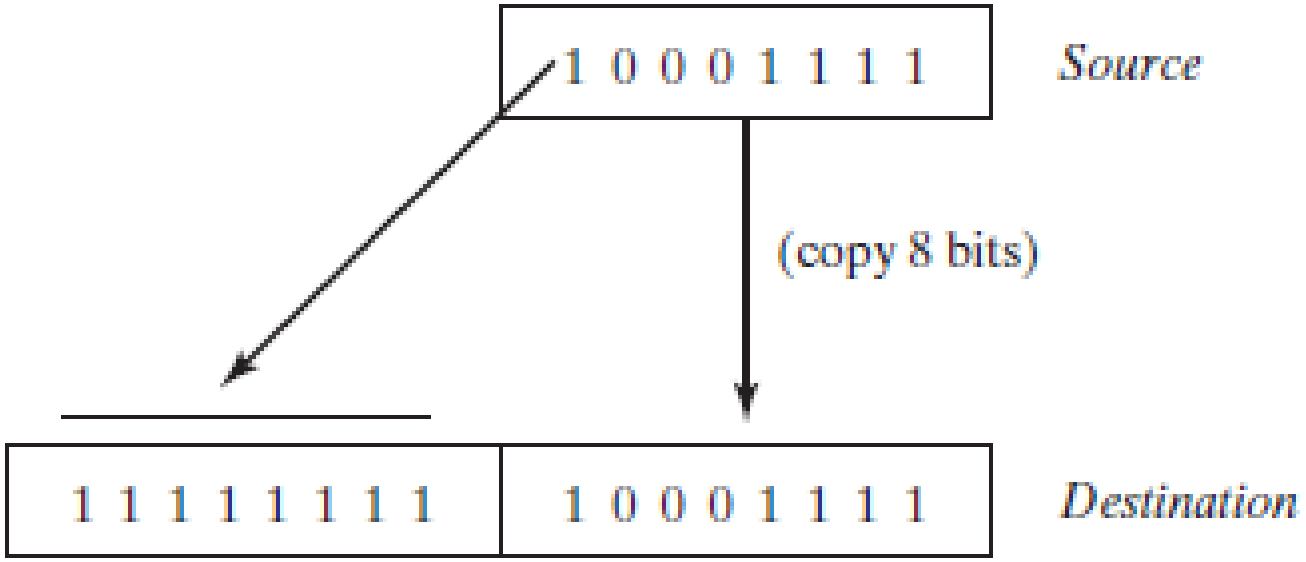
The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

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

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

```
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 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
```

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
```

Range Checking MASM has no built-in range checking for effective addresses. If we execute the following statement, the assembler just 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
```

4.2.1 INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a single 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
```

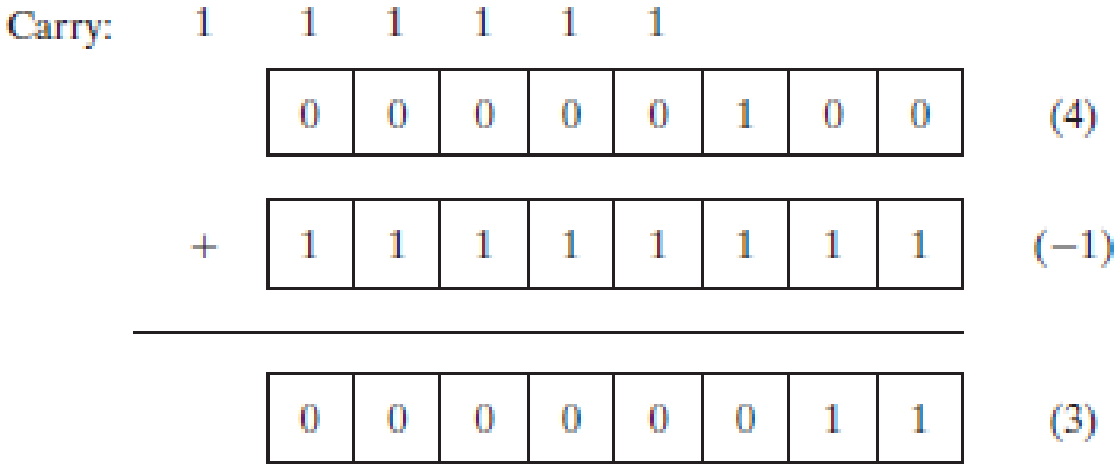
The Overflow, Sign, Zero, Auxiliary Carry, and Parity flags are changed according to the value of the destination operand. The INC and DEC instructions do not affect the Carry flag (which is something of a surprise).

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
```

Internally, the CPU can implement subtraction as a combination of negation and addition. Figure 4–3 shows how the expression 4 − 1 can be rewritten as 4 + (−1). Two's-complement notation is used for negative numbers, so −1 is represented by 11111111.

FIGURE 4–3 Adding the Value −1 to 4.



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
```

4.2.5 Implementing Arithmetic Expressions

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

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**:

```
; 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
```

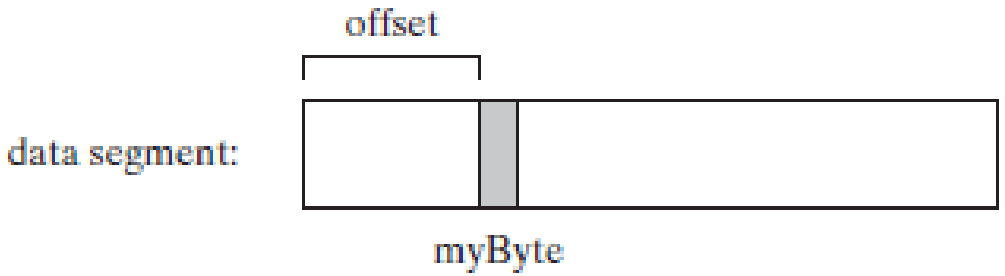
Finally, the two terms (in EAX and EBX) are added:

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

4.3.1 OFFSET Operator

The OFFSET operator returns the offset of a data label. The offset represents the distance, in bytes, of the label from the beginning of the data segment. To illustrate, Figure 4–7 shows a variable named **myByte** inside the data segment.

FIGURE 4–7 A Variable Named myByte.



OFFSET Example

In the next example, we declare three different types of variables:

```
.data
bVal  BYTE  ?
wVal  WORD  ?
dVal  DWORD ?
dVal2 DWORD ?
```

If **bVal** were located at offset 00404000 (hexadecimal), the OFFSET operator would return the following values:

```
mov esi,OFFSET bVal                    ; ESI = 00404000
mov esi,OFFSET wVal                    ; ESI = 00404001
mov esi,OFFSET dVal                    ; ESI = 00404003
mov esi,OFFSET dVal2                   ; ESI = 00404007
```

OFFSET can also be applied to a direct-offset operand. Suppose **myArray** contains five 16-bit words. The following MOV instruction obtains the offset of **myArray**, adds 4, and moves the resulting address to ESI. We can say that ESI points to the third integer in the array:

```
.data
myArray WORD 1,2,3,4,5
.code
mov  esi,OFFSET myArray + 4
```

You can initialize a doubleword variable with the offset of another variable, effectively creating a pointer. In the following example, **pArray** points to the beginning of **bigArray**:

```
.data
bigArray DWORD 500 DUP(?)
pArray DWORD bigArray
```

The following statement loads the pointer's value into ESI, so the register can point to the beginning of the array:

```
mov esi,pArray
```

4.3.2 ALIGN Directive

The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary. The syntax is

ALIGN *bound*

Bound can be 1, 2, 4, or 16. A value of 1 aligns the next variable on a 1-byte boundary (the default). If bound is 2, the next variable is aligned on an even-numbered address. If bound is 4, the next address is a multiple of 4. If bound is 16, the next address is a multiple of 16, a paragraph boundary. The assembler can insert one or more empty bytes before the variable to fix the alignment. Why bother aligning data? Because the CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses.

```
bVal   BYTE  ?                ; 00404000
ALIGN 2
wVal   WORD  ?                ; 00404002
bVal2  BYTE  ?                ; 00404004
ALIGN 4
dVal   DWORD ?                ; 00404008
dVal2  DWORD ?                ; 0040400C
```

Note that **dVal** would have been at offset 00404005, but the ALIGN 4 directive bumped it up to offset 00404008.

4.3.3 PTR Operator

You can use the PTR operator to override the declared size of an operand. This is only necessary when you're trying to access the variable using a size attribute that's different from the one used to declare the variable.

Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the operand sizes do not match:

```
.data
myDouble  DWORD  12345678h
.code
mov  ax,myDouble                    ; error
```

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

```
mov  ax,WORD PTR myDouble
mov   ax,WORD PTR [myDouble+2]          ; 1234h
```

Moving Smaller Values into Larger Destinations

```
.data
wordList WORD 5678h,1234h
.code
mov  eax,DWORD PTR wordList              ; EAX = 12345678h
```

4.3.4 TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a variable. For example, the TYPE of a byte equals 1, the TYPE of a word equals 2, the TYPE of a doubleword is 4, and the TYPE of a quadword is 8. Here are examples of each:

```
.data
var1 BYTE  ?
var2 WORD  ?
var3 DWORD ?
var4 QWORD ?
```

The following table shows the value of each TYPE expression.

Expression	Value
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.5 LENGTHOF Operator

The LENGTHOF operator counts the number of elements in an array, defined by the values appearing on the same line as its label. We will use the following data as an example:

```
.data
byte1     BYTE  10,20,30
array1    WORD  30 DUP(?),0,0
array2    WORD  5 DUP(3 DUP(?))
array3    DWORD 1,2,3,4
digitStr  BYTE  "12345678",0
```

When nested DUP operators are used in an array definition, LENGTHOF returns the product of the two counters. The following table lists the values returned by each LENGTHOF expression:

Expression	Value
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

4.3.6 SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE. In the following example, **intArray** has TYPE = 2 and LENGTHOF = 32. Therefore, SIZEOF **intArray** equals 64:

```
.data
intArray WORD 32 DUP(0)
.code
mov  eax,SIZEOF intArray          ; EAX = 64
```

4.4.2 Arrays

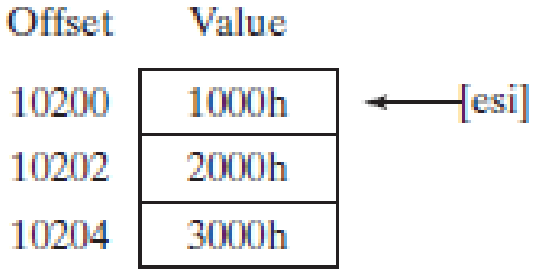
Indirect operands are ideal tools for stepping through arrays. In the next example, **arrayB** contains 3 bytes. As ESI is incremented, it points to each byte, in order:

```
.data
arrayB  BYTE 10h,20h,30h
.code
mov  esi,OFFSET arrayB
mov  al,[esi]                    ; AL = 10h
inc  esi
mov  al,[esi]                    ; AL = 20h
inc  esi
mov  al,[esi]                    ; AL = 30h
```

If we use an array of 16-bit integers, we add 2 to ESI to address each subsequent array element:

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
mov  esi,OFFSET arrayW
mov  ax,[esi]                    ; AX = 1000h
add  esi,2
mov  ax,[esi]                    ; AX = 2000h
add  esi,2
mov  ax,[esi]                    ; AX = 3000h
```

Suppose **arrayW** is located at offset 10200h. The following illustration shows the initial value of ESI in relation to the array data:



4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable's offset. Here are examples that show both notational forms:

`arrayB[esi]`	`[arrayB + esi]`
`arrayD[ebx]`	`[arrayD + ebx]`

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

```
.data
arrayB BYTE 10h,20h,30h
.code
mov  esi,0
mov  al,[arrayB + esi]          ; AL = 10h
```

The last statement adds ESI to the offset of **arrayB**. The address generated by the expression **[arrayB + ESI]** is dereferenced and the byte in memory is copied to AL.

Adding Displacements The second type of indexed addressing combines a register with a constant offset. The index register holds the base address of an array or structure, and the constant identifies offsets of various array elements. The following example shows how to do this with an array of 16-bit words:

```
.data
arrayW  WORD 1000h,2000h,3000h
.code
mov  esi,OFFSET arrayW
mov  ax,[esi]                    ; AX = 1000h
mov  ax,[esi+2]                  ; AX = 2000h
mov  ax,[esi+4]                  ; AX = 3000h
```

Scale Factors in Indexed Operands

Indexed operands must take into account the size of each array element when calculating offsets. Using an array of doublewords, as in the following example, we multiply the subscript (3) by 4 (the size of a doubleword) to generate the offset of the array element containing 400h:

```
.data
arrayD  DWORD 100h, 200h, 300h, 400h
.code
mov  esi,3 * TYPE arrayD              ; offset of arrayD[3]
mov  eax,arrayD[esi]                  ; EAX = 400h
```

```
.data
arrayD  DWORD 1,2,3,4
.code
mov  esi,3                              ; subscript
mov  eax,arrayD[esi*4]                  ; EAX = 4
```

The TYPE operator can make the indexing more flexible should arrayD be redefined as another type in the future:

```
mov  esi,3                              ; subscript
mov  eax,arrayD[esi*TYPE arrayD]        ; EAX = 4
```