Data Transfers and Arithmetic Operations

Text Two (Chapter Four)

The MOV Instruction

- The MOV instruction copies data from a source operand to a destination operand.
- This is known as a *data transfer* instruction. You will use it in almost every program you write.

CODE: MOV INSTRUCTION SYNTAX

```
MOV destination, source
```

- The right to left movement of data is similar to the following higher level programming assignment statement: *destination = source*

Rules of the MOV

- Both operands must be the same size.
- Both operands cannot be memory operands.
- CS, EIP, and IP cannot be destination operands.
- An immediate value cannot be moved to a segment register.
- Generally, the following MOV types are valid:

NOTE: VALID MOVs

MOV *reg,reg*
MOV *mem,reg*
MOV *reg,mem*
MOV *mem,imm*
MOV *reg,imm*

reg=register eg. EAX, AL
imm=immediate value eg. 56, 10h
mem=memory location eg [EAX], myvar

- Segment Registers should not be directly modified by programs running in protected mode.
- This option is available in real mode however, with the exception of the CS register.
- Generally, the following MOV types are valid, when considering segment registers.

NOTE: VALID SEGMENT REGISTER MOVs

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

sreg = segment register
mem16=16 bit memory location

SELF TEST EXERCISES

What is wrong with the following block of code?

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

Rewrite the program to do what the programmer originally intended.

Overwriting Registers with the MOV instruction

- The following code example illustrates how a 32-bit register can be modified using differently sized data.

```
.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
```

- MOV cannot directly copy data from a smaller operand to a larger one.
- Problem: Suppose *myval* stores an unsigned 16-bit value that must be copied to *ECX*.
- Solution: Set set *ECX* to zero and move *myval* to CX:

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

- Bigger Problem: What if the value of myval is a signed integer, say -16?

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

- Solution: If we set ECX to FFFFFFFFh and then copy myval to CX, the final value will be correct. (*can you verify this?*)

- Intel provides the MOVZX and MOVSX instructions to deal with these situations in general

The MOVZX instruction

- The MOVZX instruction copies the contents of a source operand into a destination operand for unsigned integers only.
- The main difference with *MOV* is the '*ZX*' component which means *Zero-eXtend.*
- It *extends* the copied value to 16 or 32 bits by automatically filling the remaining bits with *zeros*.

NOTE: VALID MOVES FOR MOVZX

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

SELF TEST EXERCISES

1. What is the value in the register AX after the following block of code executes?

```
.data
newValue BYTE 10001100b
.code
movzx ax, newValue
```

2. Rewrite the program to reduce the number of mov instructions.

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

The MOVSX instructions

- The MOVSX instruction copies the contents of a source operand into a destination operand for unsigned integers only.
- The main difference with *MOV* is the '*SX*' component which means *Sign-eXtend.*
- It *sign-extends* the copied value to 16 or 32 bits by automatically filling the remaining bits with *the highest bit of the smaller operand*.

NOTE: VALID MOVES FOR MOVSX

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

SELF TEST EXERCISES

1. What is the value in the register AX after the following block of code executes?

```
.data
byteVAl BYTE 10001100b
.code
movsx ax, byteVAl
```

2. Rewrite the program to reduce the number of mov instructions.

```
.data
count WORD FFFFh;
.code
mov ecx,FFFFFFFFh
mov cx,count
```

Other Useful Instructions

- LAHF – load status flags into AH : copies the low byte of the EFLAGS register into AH.
- SAHF - store AH into status flags : copies AH into the low byte of the EFLAGS register.
- XCHG - exchange data: exchanges the contents of two operands. Valid for *reg,reg reg,mem* and *mem,reg*

SELF TEST EXERCISES

1. Using only the mov instruction, write an ASL program that sets the values of ax and bx to 1000h and 2000h respectively and then swaps their contents.

2. Repeat the exercise without the limitation to the mov instruction. Make sure your code is shorter this time.

3. What is the content of register AH, after the following piece of code executes?

```
Newtype BYTE 1001111b;
mov ax, newtype;
add ax, newtype;
lahf;
```

SELF TEST EXERCISES

4. Can you predict the appropriate register values?

```
TITLE Data Transfer Examples (Moves.asm)
INCLUDE Irvine32.inc
.data
arrayB BYTE 10h,20h,30h,40h,50h
arrayW WORD 100h,200h,300h

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

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

SELF TEST EXERCISES

Use the following variable definitions for the remaining questions in this section:

```
.data
var1 SBYTE -4,-2,3,1
var2 WORD 1000h,2000h,3000h,4000h
var3 SWORD -16,-42
var4 DWORD 1,2,3,4,5
```

5. For each of the following statements, state whether or not the instruction is valid:

a. mov ax,var1
b. mov ax,var2
c. mov eax,var3
d. mov var2,var3
e. movzx ax,var2
f. movzx var2,al
g. mov ds,ax
h. mov ds,1000h

SELF TEST EXERCISES

6. What will be the value of the destination operand after each of the following instructions execute in sequence?

```
mov ax,var2
mov ax,[var2+4]
mov ax,var3
mov ax,[var3-2]
```

Addition and Subtraction

- Generally, for arithmetic operations, the Overflow, Sign, Zero, Auxiliary Carry, and Parity flags are changed according to the value of the destination operand.
- The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a single operand. NEG negates a number. (2's complement)

CODE: INC,DEC and NEG INSTRUCTION SYNTAX

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

- The INC and DEC instructions do not affect the Carry flag.

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

CODE: ADD INSTRUCTION SYNTAX

```
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.

- The SUB instruction subtracts a source operand from a destination operand.

CODE: SUB INSTRUCTION SYNTAX

```
SUB dest,source
```

- *Source* is unchanged by the operation, and the result is stored in the destination operand.
- The set of possible operands is the same as for the MOV and ADD instructions.

- Using ADD, SUB and NEG, it should be possible to implement mathematical expressions of addition and subtraction.

Example:

- How might a higher level language such as c, c++ or java solve an equation such as

$$value = -b + (\ c\ - a);$$

Let b, c and a be 26, 30 and 40 respectively.

Recall the EFLAGS register?

- Flags indicate the condition of the microprocessor and control its operation.
  - **Carry** flag (CF): result is too large to fit into the destination (*unsigned*).
  - **Overflow** flag (OF): result is too large or too small to fit into the destination (*signed*).
  - **Sign** flag (SF): result is negative.
  - **Zero** flag (ZF): result is zero.
  - **Auxiliary Carry** flag (AC): a carry from bit 3 to bit 4 in an 8-bit operand.
  - The **Parity** flag (PF): the least-significant byte in the result contains an even number of 1 bits.

SELF TEST EXERCISE

1. Write down the values of the indicated flags after each instruction has executed.

```
mov ax,7FF0h
add al,10h          ; CF = SF = ZF = OF =
add ah,1            ; CF = SF = ZF = OF =
add ax,2            ; CF = SF = ZF = OF =
```

Data-Related Operators and Directives

The OFFSET Operator

- The OFFSET operator returns the distance of a variable from the beginning of its enclosing segment.
- Example: Take bVal to be located at address 00404000,

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

mov esi,OFFSET bVal        ; ESI = ?
mov esi,OFFSET wVal        ; ESI = ?
mov esi,OFFSET dVal2       ; ESI = ?
```

The ALIGN directive

- The ALIGN directive aligns a *variable on a boundary using the syntax ALIGN bound.*
- *The boundary (bound) may be 1, 2, 4, or 16. Note that there is no 8.*
- *Example:*

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

- Which ASL mnemonic is similar to ALIGN? What is the difference?

The PTR Operator

- The PTR operator is used to override the declared size of an operand.
- Example:

  Will the following run without errors?

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

  What if the last line had been this:

```
mov ax,WORD PTR myDouble          ;AX=5678
```

- PTR must be used in combination with one of the standard assembler data types.

The TYPE Operator

- The TYPE operator returns the size, in bytes, of a single element of a variable.
- Example:

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

Mov eax, TYPE var1          eax=?
Mov ecx, TYPE var4          ebx=?
```

The 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.

```
.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
```

- What will lengthof return if used with each of the declared variables? i.e. byte1, array1, array2, array3 and digitStr?

SIZEOF Operator

- The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.
- Example:

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

The LABEL directive

- The LABEL directive lets you insert a label and give it a size attribute without allocating any storage.
- A common use of LABEL is to provide an alternative name and size attribute for the variable ***declared next*** in the data segment.

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov ax,val16                    ; AX = 5678h
mov dx,[val16+2]                ; DX = 1234h
```

Indirect Addressing

- Indirect addressing allows us to use a register as a pointer and manipulate the register's value.
- *Protected Mode:* In protected mode, an indirect operand can be any 32-bit general-purpose register surrounded by brackets.
- Example:

```
.data
byteVal BYTE 10h
.code
mov esi,OFFSET byteVal
mov al,[esi]                    ; AL = 10h
```

What would be the content of AL if we had written instead:

```
Mov al, esi
```

- In *real-address mode*, a 16-bit register holds the offset of a variable.
- It may be SI, DI, BX, or BP. Avoid BP unless you are using it to index into the stack.
- Example:

```
.data
byteVal BYTE 10h
.code
main PROC
startup
mov si,OFFSET byteVal
mov al,[si]                    ; AL = 10h
```

Question: What though is the actual physical address?

- Indirect operands are often used to step through arrays.
- Does the following example make sense?

```
.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
```

SELF TEST EXERCISE

1. Can you rewrite the previous program to achieve the same result if arrayB was declared as a word?

2. Can you explain how the following assembly language code works?

```
.data
arrayD DWORD 10000h,20000h,30000h
.code
mov esi,OFFSET arrayD
mov eax,[esi] ; first number
add esi,4
add eax,[esi] ; second number
add esi,4
add eax,[esi] ; third number
```

- 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.

SYNTAX: *constant*[*reg*]
[*constant* + *reg*]

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

SELF TEST EXERCISE

1. What is the content of the register al?

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

2. What is the content of the register ax?

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