Today’s Topics

• Section 4.1: Data Transfer Instructions.
• Section 4.2: Arithmetic Instructions.
Data Transfer Instructions

- MOV is for moving data between:
  - Memory
  - Register
  - Immediate (constant)
- Almost all combinations, except:
  - Memory to Memory!

MOV Instruction

- Move from source to destination. Syntax:
  
  MOV destination,source

- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

```assembly
.data
count BYTE 100
wVal WORD 2
.code
  mov bl,count
  mov ax,wVal
  mov count,al
  mov al,wVal ; error
  mov ax, count ; error
  mov eax, count ; error
```
Your turn . . .

Explain why each of the following MOV statements are invalid:

```assembly
.data
bVal BYTE 100
bVal2 BYTE ?
wVal WORD 2
dVal DWORD 5
.code
    mov ds,45 ; a.
    mov esi,wVal ; b.
    mov eip,dVal ; c.
    mov 25,bVal ; d.
    mov bVal2,bVal ; e.
```

Memory to Memory?

- Must go through a register...

```assembly
.data
Var1 WORD 100h
Var2 WORD ?
.code
    MOV AX, var1
    MOV var2, AX
```
Instruction Operand Notation

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DL, DH, DL</td>
</tr>
<tr>
<td>$r16$</td>
<td>16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP</td>
</tr>
<tr>
<td>$r32$</td>
<td>32-bit general-purpose register: EAX, EBX, ECX, EDX, ESP, EDI, ESI, EBP</td>
</tr>
<tr>
<td>$m$</td>
<td>any general-purpose register</td>
</tr>
<tr>
<td>$eseg$</td>
<td>16-bit segment register: CS, DS, SS, ES, FS, GS</td>
</tr>
<tr>
<td>$imm$</td>
<td>8-, 16-, or 32-bit immediate value</td>
</tr>
<tr>
<td>$imm8$</td>
<td>8-bit immediate byte value</td>
</tr>
<tr>
<td>$imm16$</td>
<td>16-bit immediate word value</td>
</tr>
<tr>
<td>$imm32$</td>
<td>32-bit immediate dword/word value</td>
</tr>
<tr>
<td>$im8$</td>
<td>8-bit operand which can be an 8-bit general register or memory byte</td>
</tr>
<tr>
<td>$im16$</td>
<td>16-bit operand which can be a 16-bit general register or memory word</td>
</tr>
<tr>
<td>$im32$</td>
<td>32-bit operand which can be a 32-bit general register or memory doubleword</td>
</tr>
<tr>
<td>$mem$</td>
<td>an 8-, 16-, or 32-bit memory operand</td>
</tr>
</tbody>
</table>

For examples, see the ADC and ADD instructions in Appendix B.2 (p. 620)

Direct Memory Operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h
.code
mov al,var1       ; AL = 10h
mov al,[00010400]  ; if var1 at offset 10400h
mov al,[var1]      ; AL = 10h
```
Zero or Sign Extension

• What happens to ECX if –1 is moved to CX?
  – Are the higher 16 bits of ECX all 0?
  – What number does ECX represent now?
• The solution: MOVZX and MOVSX
  – MOVZX always fills higher bits with 0.
  – MOVSX fills higher bits by “sign extension”.
  – Just extend the left-most bit!

Zero Extension

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.

\[
\begin{array}{c}
\text{Source} \\
\downarrow \\
\text{Destination}
\end{array}
\]

\text{mov bl,}10001111b
\text{movzx ax,bl ; zero-extension}

The destination must be a register.
Sign Extension

The MOVsx instruction fills the upper half of the destination with a copy of the source operand’s sign bit.

![Diagram showing sign extension](image)

```
mov bl,10001111b
movsx ax,bl ; sign extension
```

The destination must be a register.

LAHF/SAHF and XCHG

- LAHF to load flags into AH.
- SAHF to go the other way.

- XCHG for exchange data between:
  - Register, register
  - Register, memory
  - Memory, register
  (again, no memory to memory)
Direct-Offset Operands

• Adding a displacement (or offset) to a variable name:

```
arrayB BYTE 10h, 20h, 30h, 40h, 50h
...
MOV AL, arrayB ; AL=10h
MOV AL, [arrayB+1] ; AL=20h
MOV AL, arrayB+1 ; Is it valid?
; Hint: Can you do an addition without an ADD instruction?
```

Addition and Subtraction

• ADD X, Y
  \( X := X + Y \)

• SUB X, Y
  \( X := X - Y \)
INC, DEC, NEG

- **INC X**
  \[ X := X + 1 \text{ or } X++ \]
- **DEC X**
  \[ X := X - 1 \text{ or } X-- \]
- **NEG X**
  \[ X := -X \]

Expression

- Example: \( X=(A + B) \times (D - E) \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register-Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>EAX, A</td>
<td>Load A to EAX</td>
</tr>
<tr>
<td>ADD</td>
<td>EAX, B</td>
<td>Add B to EAX</td>
</tr>
<tr>
<td>MOV</td>
<td>ECX, D</td>
<td>Load D to ECX</td>
</tr>
<tr>
<td>SUB</td>
<td>ECX, E</td>
<td>Subtract E from ECX</td>
</tr>
<tr>
<td>IMUL</td>
<td>EAX, ECX</td>
<td>Multiply EAX by ECX</td>
</tr>
<tr>
<td>MOV</td>
<td>X, EAX</td>
<td>Move EAX to X</td>
</tr>
</tbody>
</table>
Flags Affected

• Flags (register) tell us whether any of the following conditions occur:
  – Overflow,
  – Carry,
  – Zero, Sign…etc.
• Used for decision in branch.
  – Loop (discussed next)
  – If…then…else

Zero and Sign

• Zero Flag ZF=1 if the instruction produce 0.
  MOV CX, 1
  SUB CX, 1 ; CX=0, ZF=1
• Sign Flag SF=1 if the instruction produce a negative number.
  MOV CX, 0
  SUB CX, 1 ; CX=-1, SF=1
  ADD CX, 2 ; CX=1, SF=0
Carry (Unsigned Arithmetic)

- Example:
  MOV AL, 0FFh
  ADD AL, 1 ; CF = 1, AL=00
  MOV AX, 00FFh
  ADD AX, 1 ; CF = 0, AX=0100h

Overflow (Signed Arithmetic)

- Example:
  MOV AL, +127
  ADD AL, 1 ; OF = 1
  MOV AL, -128
  SUB AL, 1 ; OF = 1
Detecting Carry

• Detecting Carry is easy.
  – Adding two N-bit numbers result in an (N+1)-bit number.
• Example:
  \[
  \begin{array}{cccccccc}
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  + & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
  \end{array}
  \]
• CF is ignored for signed arithmetic. For example, the above is 4 + (-1) in decimal

Detecting Overflow

• Carry isn’t meaningful for signed arithmetic. For example, adding any two negative numbers always produces carry.
• Detecting Overflow:
  – Compare CF and the bit carried into MSB (Most Significant Bit).
  – See textbook page 112 for more detail.
Overflow in Positive Numbers

- Carry never happens. For example:
  \[
  0 1 1 1 1 1 1 1 \quad (127) \\
  + 0 1 1 1 1 1 1 1 \quad (127)
  \]
- Overflow occurs if MSB becomes 1
  - MSB=1 indicates a negative number.
  - But, we’re adding two positive numbers…?! 

Overflow in Negative Numbers

- Carry always happens. For example:
  \[
  1 0 0 0 0 0 0 0 \quad (-128) \\
  + 1 1 1 1 1 1 1 1 \quad (-1)
  \]
- Overflow occurs if MSB becomes 0
  - MSB=0 indicates a positive number.
  - But, we’re adding two negative numbers…?!
Detecting Overflow

• Overflow: $CF \neq MSB$?
  – Doesn’t work if adding a positive number to a negative number (or vice versa)!

• Overflow: ($CF \neq MSB$) and not the case of (positive+negative)
  – Overflow: ($CF \neq MSB$) and (MSB before carry-in is 0)?
  – A better result: Overflow: $CF \neq$ (carry-in to MSB)