Computer Organization &
Assembly Language Programming

CSE 2312
Lecture 25 Summary of Basic Instructions

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Assembly Language: Basic Instructions

• Instruction Groups
  – Data Transfer Instructions
  – Addition and Subtraction
  – Multiplication and Division
  – Data-Related Operators and Directives
  – JMP and LOOP Instructions
### 8088 Instruction Set (1)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Operands</th>
<th>Status flags</th>
<th>O</th>
<th>S</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>Move word, byte</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCHG(B)</td>
<td>Exchange word</td>
<td>( r \leftrightarrow e )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
<td>( r \leftarrow #e )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
<td>( e, # )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>Pop from stack</td>
<td>( e )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XLAT</td>
<td>Translate AL</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD(B)</td>
<td>Add word</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC(B)</td>
<td>Add word with carry</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB(B)</td>
<td>Subtract word</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB(B)</td>
<td>Subtract word with borrow</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL(B)</td>
<td>Multiply signed</td>
<td>( e )</td>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL(B)</td>
<td>Multiply unsigned</td>
<td>( e )</td>
<td>U</td>
<td></td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDIV(B)</td>
<td>Divide signed</td>
<td>( e )</td>
<td>U</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>DIV(B)</td>
<td>Divide unsigned</td>
<td>( e )</td>
<td>U</td>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

- Data Transfer Instructions
- Addition and Subtraction
- Multiplication and Division
### 8088 Instruction Set (2)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Operands</th>
<th>Status flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>CBW</td>
<td>Sign extend byte-word</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CWD</td>
<td>Sign extend word-double</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NEG(B)</td>
<td>Negate binary</td>
<td>e</td>
<td>*</td>
</tr>
<tr>
<td>NOT(B)</td>
<td>Logical complement</td>
<td>e</td>
<td>-</td>
</tr>
<tr>
<td>INC(B)</td>
<td>Increment destination</td>
<td>e</td>
<td>*</td>
</tr>
<tr>
<td>DEC(B)</td>
<td>Decrement destination</td>
<td>e</td>
<td>*</td>
</tr>
<tr>
<td>AND(B)</td>
<td>Logical and</td>
<td>e ← r, r ← e, e ← #</td>
<td>0</td>
</tr>
<tr>
<td>OR(B)</td>
<td>Logical or</td>
<td>e ← r, r ← e, e ← #</td>
<td>0</td>
</tr>
<tr>
<td>XOR(B)</td>
<td>Logical exclusive or</td>
<td>e ← r, r ← e, e ← #</td>
<td>0</td>
</tr>
<tr>
<td>SHR(B)</td>
<td>Logical shift right</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>SAR(B)</td>
<td>Arithmetic shift right</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>SAL(B) (=SHL(B))</td>
<td>shift left</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>ROL(B)</td>
<td>Rotate left</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>ROR(B)</td>
<td>Rotate right</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>RCL(B)</td>
<td>Rotate left with carry</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
<tr>
<td>RCR(B)</td>
<td>Rotate right with carry</td>
<td>e ← 1, e ← CL</td>
<td>*</td>
</tr>
</tbody>
</table>

- Data-Related Operators and Directives
### 8088 Instruction Set (3)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Operands</th>
<th>Status flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST(B)</td>
<td>Test operands</td>
<td>e ← r, e ← #</td>
<td>0</td>
</tr>
<tr>
<td>CMP(B)</td>
<td>Compare operands</td>
<td>e ← r, e ← #</td>
<td>*</td>
</tr>
<tr>
<td>STD</td>
<td>Set direction flag (↓)</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear direction flag (↑)</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>STC</td>
<td>Set carry flag</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry flag</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>CMC</td>
<td>Complement carry</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>LOOP</td>
<td>Jump back if decremented CX ≥ 0</td>
<td>label</td>
<td>-</td>
</tr>
<tr>
<td>LOOPZ LOOPZE</td>
<td>Back if Z=1 and DEC(CX)≥0</td>
<td>label</td>
<td>-</td>
</tr>
<tr>
<td>LOOPNZ LOOPNZE</td>
<td>Back if Z=0 and DEC(CX)≥0</td>
<td>label</td>
<td>-</td>
</tr>
<tr>
<td>REP REPZ REPNZ</td>
<td>Repeat string instruction</td>
<td>string instruction</td>
<td>-</td>
</tr>
<tr>
<td>MOVES(B)</td>
<td>Move word string</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LODS(B)</td>
<td>Load word string</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>STOS(B)</td>
<td>Store word string</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SCAS(B)</td>
<td>Scan word string</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CMPS(B)</td>
<td>Compare word string</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JCC</td>
<td>Jump according conditions</td>
<td>label</td>
<td>-</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump to label</td>
<td>e, label</td>
<td>-</td>
</tr>
<tr>
<td>CALL</td>
<td>Jump to subroutine</td>
<td>e, label</td>
<td>-</td>
</tr>
<tr>
<td>RET</td>
<td>Return from subroutine</td>
<td>#</td>
<td>-</td>
</tr>
<tr>
<td>SYS</td>
<td>System call trap</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- JMP and LOOP Instructions
Data Transfer Instructions

- **Data Transfer Instructions**
  - MOV or MOVB
  - XCHG or XCHGB
  - PUSH or POP
  - PUSHF or POPF
  - XLAT

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Operands</th>
<th>Status flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>Move word, byte</td>
<td>( r \leftarrow e, e \leftarrow r, e \leftarrow # )</td>
<td>-</td>
</tr>
<tr>
<td>XCHG(B)</td>
<td>Exchange word</td>
<td>( r \leftarrow e )</td>
<td>-</td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
<td>( r \leftarrow #e )</td>
<td>-</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
<td>( e, # )</td>
<td>-</td>
</tr>
<tr>
<td>POP</td>
<td>Pop from stack</td>
<td>( e )</td>
<td>-</td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XLAT</td>
<td>Translate AL</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Move Instructions

• Copy and Move Instructions
  – Move from source to destination. Syntax: MOV destination, source
  – If the source is a register, the destination can be an effective address. In this table a register operand is indicated by an \( r \) and an effective address by an \( e \), so this operand combination is denoted by \( e \leftarrow r \).
  – Since, in the instruction syntax, the destination is the first operand and the source is the second operand, the arrow \( \leftarrow \) is used to indicate the operands. Thus, \( e \leftarrow r \) means that a register is copied to an effective address.
  – For the MOV instruction, the source can also be an effective address and the destination a register, which will be denoted by \( r \leftarrow e \).
  – Another choice is immediate data as source, and effective address as destination, which yields \( e \leftarrow \# \). Immediate data in the table is indicated by the sharp sign (\#).
  – The Move instructions do not move data. They make copies, meaning that the source is not modified as would happen with a true move.

• MOV or MOVB
  – word move MOV and the byte move MOVB
  – No more than one memory operand permitted
  – CS, EIP, and IP cannot be the destination
  – No immediate to segment moves
Your Turn

- **MOV DS, 45**
  - Explain why it is invalid?
  - Immediate move to DS not permitted
- **MOV 25, VAR**
  - Explain why it is invalid?
  - Immediate value cannot be destination
- **MOV EIP, VAR**
  - Explain why it is invalid?
  - EIP can not be destination
- **MOV AX, BH**
  - Explain why it is invalid?
  - The sizes are missed to match
XCHG Instruction

• **XCHG**
  – Exchanges the contents of a register with the contents of an effective address.
  – For the exchange, the table uses the symbol ↔. In this case, there exists a byte version as well as a word version.
  – Thus, the instruction is denoted by XCHG and the Operand field contains $r \leftrightarrow e$.
  – At least one operand must be a register.
  – No immediate operands are permitted.

  – Example:
    ```assembly
    XCHG AX, BX   ! Exchange 16-bit registers
    XCHGB AH, AL  ! Exchange 8-bit registers
    XCHG VAR1, BX ! Exchange memory and register
    XCHG VAR1, VAR2 ! Error: two memory operands
    ```
Your Turn

• Problem
  – Suppose AX=1, BX=2, CX=3; Write a program that rearranges their values such that AX=3; BX=1; CX=2.

• Solution
  – Step1: Exchange the values in AX and CX.
    \[
    \text{XCHG AX, CX}
    \]
  – Step2: Exchange the values in BX and CX.
    \[
    \text{XCHG BX, CX}
    \]
PUSH and POP

• **PUSH**
  – PUSH pushes its operand onto the stack.
  – The explicit operand can either be a constant (♯ in the Operands column) or an effective address (e in the Operands column).
  – There is also an implicit operand, SP, which is not mentioned in the instruction syntax.
  – What the instruction does is decrement SP by 2, then store the operand at the location now pointed to by SP.

• **POP**
  – POP removes an operand from the stack to an effective address.
  – What the instruction does is increment SP by 2, then store the operand at the location now pointed to by SP.

• **PUSHF and POPF**
  – They have implied operands, the push and pop the flags register, respectively.
  – This is also the case for XLAT which loads the byte register AL from the address computed from AL + BX. This instruction allows for rapid lookup in tables of size 256 bytes.
ADD and SUB Instructions

• ADD and SUB
  – ADD destination, source; Logic: destination ← destination + source
  – SUB destination, source; Logic: destination ← destination – source
• Operand Rules
  – Each of these has the same three operand combinations as MOV
  – Effective address to register, register to effective address, and constant to effective address. Thus, the Operands column of the table contains \( r\leftarrow e \), \( e\leftarrow r \), and \( r\leftarrow \# \).
  – In them, the overflow flag, \( O \), the sign flag, \( S \), the zero flag, \( Z \), and the carry flag, \( C \) are all set, based on the result of the instruction.
  – For example, that \( O \) is set if the result cannot be correctly expressed in the allowed number of bits, and cleared if it can be.
  – When the largest 16-bit number, 0x7fff (32,767 in decimal), is added to itself, the result cannot be expressed as a 16-bit signed number, so \( O \) is set to indicate the error.
  – For all additions and subtractions, byte versions also exist.
Example

- Problem

```assembly
.SECT .DATA
VAR1 WORD 1000h
VAR2 WORD 2000h

.SECT .TEXT
    MOV AX,VAR1
    ADD AX,VAR2
    ADD AX,0FFFh
    ADD AX,1
```

---AX---  |   1000h
---AX---  |   3000h
---AX---  |   3FFFh
---AX---  |   4000h
MUL and DIV Instructions

- **Operand Rules**
  - Unsigned integer operands require the MUL and DIV instructions;
  - AH:AL combination is the implied destination in the byte version.
  - DX:AX combination is the implied destination in the word version.
  - Even if the result is only a word/byte, the DX/AH register is rewritten.

- **Multiplication**
  - It is always possible because the destination contains enough bits.
  - The overflow/carry bits are set if the product is over one word/byte.
  - The zero and and negative flags are undefined after a multiply.

- **Division**
  - Uses the register combinations DX:AX or AH:AL as the destination.
  - The quotient goes into AX or AL and the remainder into DX or AH.
  - All four flags, carry, overflow, zero and negative, are undefined after a divide operation.
  - If the divisor is 0, it executes a trap to stops the program unless a trap handler routine is present.
  - It is sensible to handle minus signs before and after the divide, because in the 8088 definition the sign of the remainder equals the sign of the dividend, whereas in mathematics, a remainder is always nonnegative.
MUL Instruction

• **MUL**
  – The MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

• **Format**
  – MUL r/m8
  – MUL r/m16
  – MUL r/m32

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>reg/mem8</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>reg/mem16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>EAX</td>
<td>reg/mem32</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>
MUL Example

• 100h * 2000h, using 16-bit operands

```assembly
.SECT .DATA
VAL1  WORD 2000h
VAL2  WORD 100h
.SECT .TEXT

MOV AX,VAL1  ! MUL VAL2 ! DX:AX = 00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

• 12345h * 1000h, using 32-bit operands

```assembly
MOV EAX,12345h   ! MOV EBX,1000h    ! MUL EBX          !   EDX:EAX = 0000000012345000h, CF=0
MOV EBX,1000h     !
MUL EBX          !   EDX:EAX = 0000000012345000h, CF=0
```

MUL Example
Your Turn

- What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
MOV AX, 1234h
MOV BX, 100h
MUL BX
```

\[ DX = 0012h, \ AX = 3400h, \ CF = 1 \]
DIV Instruction

- **DIV**
  - The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
  - A single operand is supplied (register or memory operand), which is assumed to be the divisor
  - Instruction formats:
    - `DIV reg/mem8`
    - `DIV reg/mem16`
    - `DIV reg/mem32`

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>reg/mem8</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>reg/mem16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>EAX</td>
<td>reg/mem32</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>
DIV Example

• Divide 8003h by 100h, using 16-bit operands:

    MOV DX, 0                       ! Clear, high
    MOV AX, 8003h                   ! dividend, low
    MOV CX, 100h                    ! divisor
    DIV CX                           ! AX = 0080h, DX = 3

• Same division, using 32-bit operands:

    MOV EDX, 0                       ! Clear, high
    MOV EAX, 8003h                   ! dividend, low
    MOV ECX, 100h                    ! divisor
    DIV ECX                           ! EAX = 00000080h, EDX = 3
Your Turn

• What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
MOV DX, 0087h
MOV AX, 6000h
MOV BX, 100h
DIV BX
```

\[ DX = 0000h, \ AX = 0060h \]
Your Turn

• What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

  MOV DX,0087h
  MOV AX,6002h
  MOV BX,10h
  DIV BX

Divide Overflow